البرمجة بلغة الكينونة

Object Oriented Programming (OOP)

الدكتور
زياد عبد الكريم القاضي



مكتبة المجتمع العربي للنشر والتوزيع

البرمجة بلغة الكينونة

Object Oriented Programming(OOP)

البرمجة بلغة الكينونة

Object Oriented Programming (OOP)

تأليف

الدكتور

زياد عبد الكريم القاضي

الطبعة الأولى

2013م – 1434 هـ

مكتبة المجتمع العربي

مكتبة المجتمع العربي للنشر والتوزيع

رقم الإيداع لدى دائرة المكتبة الوطنية (2011/7/2862)

005.1

القاضي، زياد عبد الكريم

البرمجة بلغة الكينونة/زياد عبد الكريم القاضي.- عمان: مكتبة المجتمع العربي للنشر والتوزيع، 2011

(   ) ص

ر.إ. : 2011/7/2862

الواصفات: /برمجة الحاسوب//لغات الحواسيب//الحاسوب/

- يتحمل المؤلف كامل المسؤولية القانونية عن محتوى مصنفه ولا يعبر هذا المصنف عن رأي دائرة المكتبة الوطنية أو أي جهة حكومية أخرى.

الطبعة العربية الأولى

2013م – 1434هـ

مكتبة المجتمع العربي للنشر والتوزيع

عمان – وسط البلد – ش. السلط – مجمع الفحيص التجاري

تلفاكس 4632739 ص.ب. 8244 عمان 11121 الأردن

عمان – ش. الملكة رانيا العبد الله – مقابل كلية الزراعة –

مجمع زهدي حصوة التجاري

www: muj-arabi-pub.com

Email: Moj_pub@hotmail.com

ISBN 978-9957-83-116-5 (ردمك)

المحتويا ت

المقدمة

لا بد انك قد اطلعت على آلية استخدام لغة سي بلس بلس في البرمجة ولا بد انك قد كتبت ونفذت برنامجا اجاءيا بلغة سي وعليه وحتى تكون هناك فائدة من استخدام هذا الكتاب فلا بد من ان تكون قد درست البرمجة الكلاسيكية باستخدام لغة سي وان تكون عندك المعلومات الكافية لكتابة البرنامج ونخص بالذكر:

- التعامل مع انوع البيانات المختلفة.
- معرفة عمليات الادخال والاخراج
- الالمام بعمليات نقل التحكم في البرنامج من خلال استخدام الجمل الشرطية وجمل التكرار المختلفة.
- القدرة على معالجة المصفوفات.
- التعامل مع الاقترانات المختلفة والالمام بالية تمرير البيانات بين البرامج الفرعية والبرنامج الرئيس.

يعتبر هذا الكتاب مكملا لاي كتب خاص بالبرمجة بلغة سي بلس بلس وقد اقتصرنا في هذا الكتاب على شرح بعض المزايا الخاصة ببرمجة الكيانات الموجهه لما لهذه الميزة من حسنات كبيرة في تطوير البرنامج املين ان نكون قد اوصلنا هذه الفكرة بطريقة سهلة وميسرة.

والله ولي التوفيق

المؤلف

الوحدة الأولى

1

مقدمة الى المؤشرات

Introduction to pointers

مقدمة الى المؤشرات

Introduction to pointers

المؤشر ما هو الا موقع وله أهمية كبيرة في البرمجة للاسباب التالية:

- تزود المؤشرات المبرمج بطريقة قوية ومرنة للبرمجة.
- بعض اجزاء البرنامج يمكن ان تنفذ بطريقة فعالة باستخدام المؤشرات.
- لفهم عملية التعامل مع المؤشرات لا بد للمبرمج من معرفة آلية تخزين البيانات في الذاكرة.
- تتكون الذاكرة من مجموعة من المواقع بحيث تعطى ارقاما تسلسلية وبدءا من الصفر وتسمى كل قيمة بالعنوان ويمكن ان تخزن في العنوان الواحد في الذاكرة مجموعة من البتتات تسمى الكلمة.
- تتراوح قيم العناوين في نظام الكمبيوتر من الصفر الى عدد محدد يعتمد على عدد الاسلاك المخصصة لنقل العنوان (ناقل العنوان).
- يمكن الرجوع الى البيانات المخزنة في الذاكرة او تخزين بينات في الذاكرة من خلال استخدام العناوين. والشكل التالي يبين كيفية تخزين البيانات في الذاكرة وفي عناونين او مواقع مختلفة:

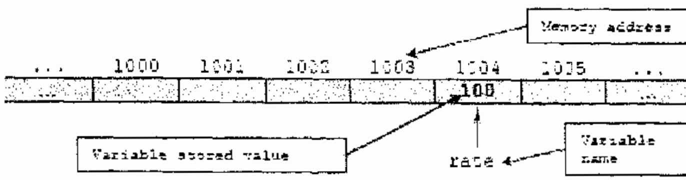
Address		Data	
00010000 00000000	= 4096	00000000 00000000	
00010000 00000010	= 4098	00000000 00001100	= 12
00010000 00000100	= 4100	00000000 00001110	= 14
00010000 00000110	= 4102	00000000 00010000	= 16
00010000 00001000	= 4104	00011000 00000000	= 6144
00010000 00001010	= 4106	00011100 00000000	= 7168

00011000 00000000	= 6144	01100011 01101100	= 99, 108 = c, l
00011000 00000010	= 6146	01110101 01100010	= 117, 98 = u, b
00011000 00000100	= 6148	00000000 00000000	= 0
00011000 00000110	= 6150	00000000 00000010	= 2 = poor
00011000 00001000	= 6152		

لناخذ جملة الاعلان التالية عن متغير رقمي:

int rate = 100;

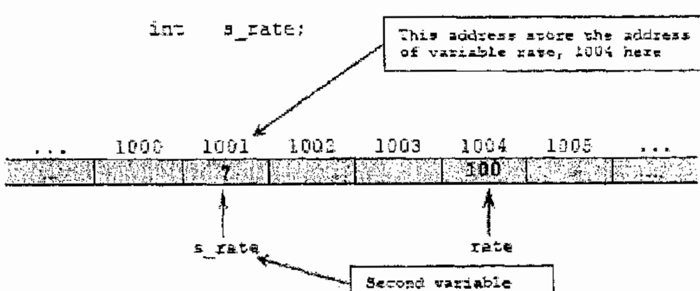
يؤدي تنفيذ هذه الجملة الى تخزين قيمة صحيحة في موقع (او أكثر) في الذاكرة وكما هو مبين في الشكل التالي:



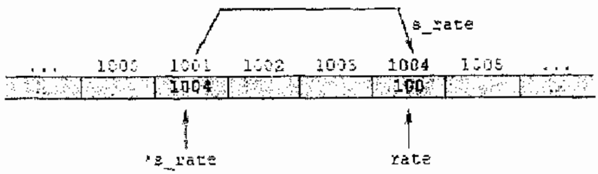
لاحظ من الشكل ان اسم المتغير ما هو إلا اسم لموقع وقيمة هذا الموقع في الشكل هي 1004.

لنعلن الان عن متغير اخر كما يلي:

int s_rate;



لنخزن الان موقع المتغير الاول في موقع المتغير الثاني كما يلي:



وبطريق مبسطة للتمثيل تظهر الذاكرة كما يلي:



Where:

address	variable name	hold data

تستخدم النجمة في السي بلس بلس للاعلان عن موقع او مؤشر كما يلي:

```
int *s_rate;
```

هذا ويمكن للمؤشر ان يشير الى انواع مختلفة من البيانات في الذاكرة مثل النوع الرمزي والصحيح والكسري وغيرها وكما هو مبين في المثال التالي:

```
char*   x;

int  *   type_of_car;

float   *value;
```

```
// ch1 and ch2 both are pointers to type char.
char   *ch1, *ch2;


// value is a pointer to type float, and percent
is an ordinary float variable.

float   *value, percent;
```

عند الاعلان عن المؤشر لا بد من تهيئته بحيث يشير الى نوع من البيانات المراد التعامل معها في البرنامج وكما هو موضح في البرنامج التالي:

لاحظ ان:

1. **Indirection operator** (*)
2. **Address-of-operator** (&) – means return the address of.

```
#include <stdio.h>


int main(void)

{

    int    *m;

    int location = 200;

                        m = &location;

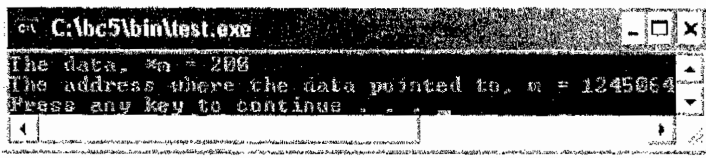
    cout<<"The data, *<<m;

    cout<<"The address where the data pointed to, m =
    "<< &m;

    return 0;

}
```

Output:



وعليه فإنه لتهيئة المؤشر تستخدم إشارة والمنطقية وكما هو مبين في المثال التالي:

```
// declare a pointer variable, m of type int
int   *m;
// assign the address of variable location

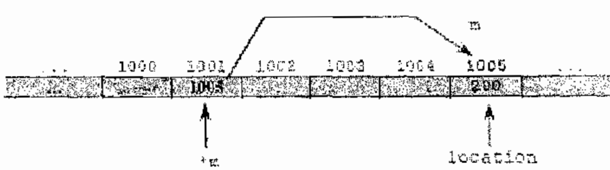
// to variable m, so pointer m is pointing to
variable location

m = &location;

// the actual data assigned to variable location

location = 200;
```

ويمكن تمثيل هذا بالرسم كما يلي:



معامل النجمة غير المباشر هو مكمل للمعامل المثل باشارة والمنطقية.
لنفحص الان التعليمات التالية:

```
m = &location;

location = 200;

q = *m;
```

التعليمة ;q = *m ستقوم بوضع قيمة البيانات الفعلية في المتغير q والذي بدوره يعني ان هذا المتغير سوف يستقبل البيانات المعنونة بالعنوان المخزن في المتغير m.

تستخدم المؤشرات وتعالج بطرق مختلفة وكما هو الحال عند التعامل مع المتغيرات فانه يمكن استخدام المؤشر في الطرف الأيمن للتعبير او لحملة المساواة. ذلك لتخصيص قيمة هذا المؤشر لمؤشر اخر.

لناخذ المثال التالي:

```
// program to illustrate the basi c use of pointers
#include <iostream>
using namespace std;

void main()
{
   // declares an integer variable and two pointers variables
   int    num = 10, *point_one, *point_two;
   // assigns the address of variable num to pointer point_one
   point_one = &num;
    // assigns the (address) point_one to point_two
    point_two = point_one;
   cout<<"Pointers variables..."<<endl;
   cout<<"*point_one = "<<*point_one<<"\n";
```

```
    cout<<"*point_two = "<<*point_two<<"\n";

    cout<<"\nNormal variable..."<<endl;

    cout<<"num = "<<num<<"\n";

    // displays value 10 stored in num since point_one

    // and point_two now point to variable num

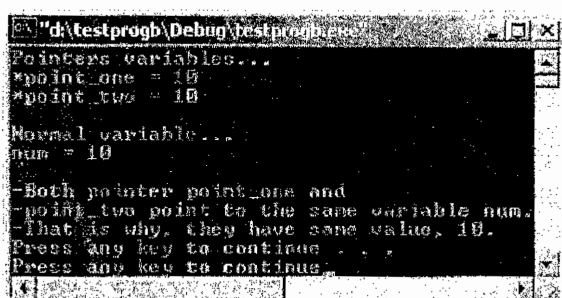
    cout<<"\n-Both pointer point_one and"<<"\n";

    cout<<"-point_two point to the same variable
num."<<"\n";

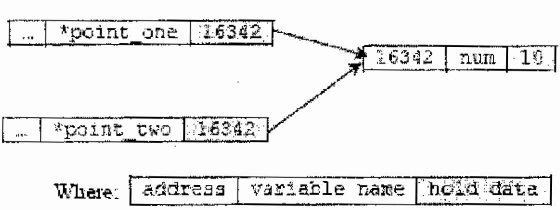
    cout<<"-That is why, they have same value, 10."<<endl;

}
```

Output:



يمكن تمثيل عمل هذا البرنامج بالرسم كما يلي:



لاحظ الفرق بين المؤشر والذي يشير الى موقع في الذاكرة والبيانات.

لاحظ من المثال السابق ما يلي:

- إمكانية الوصول الى البيانات باستخدام اسم المتغير او ما يسمى الوصول المباشر الى البيانات.
- الوصول غير المباشر الى البيانات باستخدام المؤشر.

لاحظ من التعليمات التالية ان *pter and var كلاهما يشير الى نفس المحتوى ألا وهو المتغير وكلاهما يمثل نفس عنوان البيانات المخزنة في الموقع:

```
// declare a pointer variable named pter, where
the
// data stored pointed to by pter is int type
int    *pter;
// assign the address of variable named var to a
pointer variable named pter
pter = &var;
```

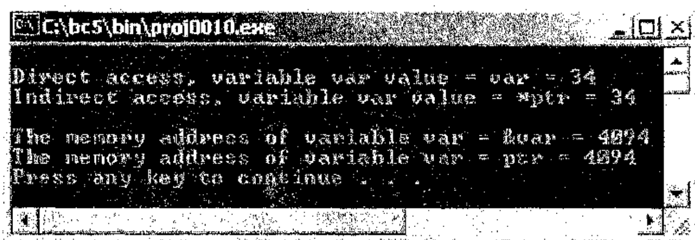
لنستعرض الان المثال التالي:

```
// a basic pointer use
#include <stdio.h>

void main()
{
    // declare and initialize an int variable
    int  var = 34;
    // declare a pointer to int variable
    int  *ptr;
    // initialize ptr to point to variable var
    ptr = &var;
    // access var directly and indirectly
    Cout<<"\nDirect access, variable var value = var ="<<
var;
```

```
    Cout<<"\nIndirect access, variable var value = *ptr ="
<< *ptr;
    // display the address of var two ways
    Cout<<"\n\nThe memory address of variable var = &var
= "<< &var;
    Cout<<"\nThe memory address of variable var = ptr
="<< ptr;
}
```

Output:



لاحظ انه اذا نفذت هذه التعليمات يمكن ان تحصل على قيم اخرى للعناوين لكن هذا لا يهمنا لاننا نتعامل مع المؤشر والذي بدوره يحول الى عنوان من قبل الحاسوب عند تنفيذ البرنامج.

لناخذ عملية الاعلان التالية:

```
int    age = 25;
```

عندها وبعد تنفيذ التعليمات التالية:

```
int    *ptr_age;
ptr_age = &age;
ptr_age++;
```

فانه اذا كانت قيمة المؤشر الصحيح 1000 فانه بعد عملية الزيادة سيصبح مساويا 1002 وذلك لانه يتم تخصيص 2 بايت للقيمة الصحيحة و4 بايت للقيمة الكسرية:

int = 2 byte.

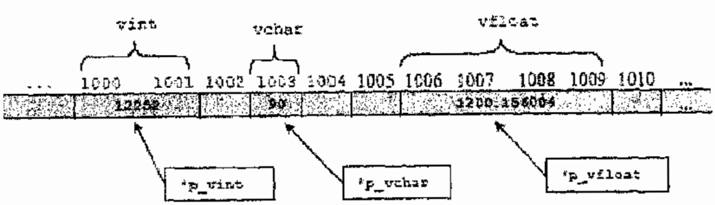
float = 4 byte.

وفي كل مرة يزاد فيها المؤشر فانه يزاد ليشير الى القيمة الصحيحة التالية:

لاحظ ما يلي:

```
int     vint = 12252;
char    vchar = 90;
float   vfloat = 1200.156004;
```

تخزن المتغيرات في الذاكرة كما هو مبين في الشكل ادناه:



وعليه فانه:

- يتم حجز 2 بايت للمتغير الصحيح (وفي بعض نماذج سي 4 بايت).
- يتم حجز بايت واحد للمتغير الرمزي.
- يتم حجز 4 بايت للمتغير الكسري.

اما التعليمات التالية فانها تعلن عن موشرات لقيم مختلفة في النوع:

```
int      *p_vint;
char     *p_vchar;
float    *p_vfloat;
```

ويمكن تهيئة هذه المؤشرات كما يلي:

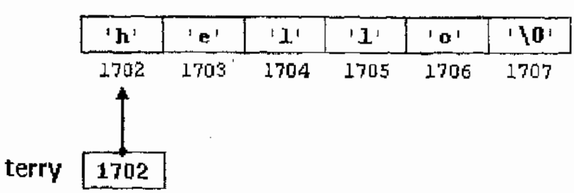
```
p_vint = &vint;
p_vchar = &vchar;
p_vfloat = &vfloat;
```

وبافتراض القيم في الشكل السابق فان قيم هذه المؤشرات هي كما يلي:

```
p_vint    equals   1000.
p_vchar   equals   1003.
p_vfloat  equals   1006.
```

لاحظ المثال التالي:

```
char * terry = "hello";
```

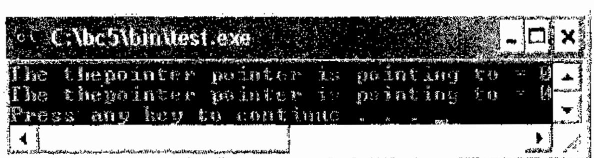


اذا لم يهيئ الموشر او اعطي قيمة نل فيمكن اعتبار القيمة صفرية او غير معروفة كما هو مبين في المثال التالي:

```
#include <stdio.h>

int main()
{
   int    *thepointer = NULL;
   // do some testing....
    Cout<<"The thepointer pointer is pointing to = "<<
thepointer;
    printf("The thepointer pointer is pointing to = "<<
thepointer;
   return 0;
}
```

Output:



تنفذ على المؤشرات عمليات متنوعة ولبيان هذا لنأخذ الاعلان التالي:

```
char *mychar;
short *myshort;
long *mylong;
```

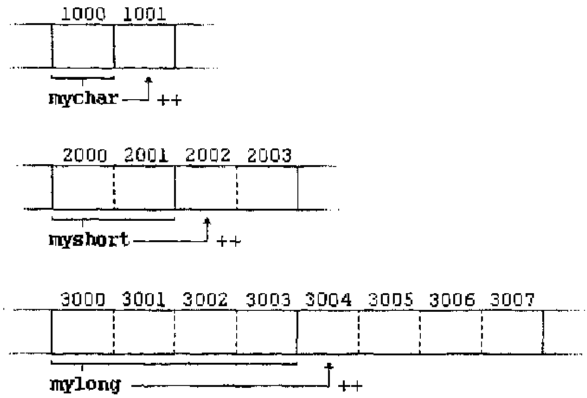
وعليه فانه اذا استخدمنا التعليمات التالية:

```
mychar++;

myshort++;

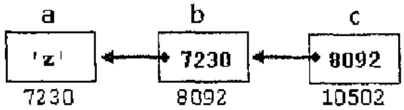
mylong++;
```

فإن المؤشرات ستزاد كما هو مبين في الشكل ادناه:



يمكن ان يشير المؤشر الى مؤشر ولبيان هذا خذ التعريف التالي:

```
char a;
char * b;
char ** c;
a = 'z';
b = &a;
c = &b;
```



وفيما بعض البرامج والتي تبين كيفية التعامل مع المؤشرات:

```
// pointer to functions
#include <iostream>
using namespace std;

int addition (int a, int b)
{ return (a+b); }
```

```
int subtraction (int a, int b)
{ return (a-b); }

int operation (int x, int y, int (*functocall)(int,int))
{
 int g;
 g = (*functocall)(x,y);
 return (g);
}

int main ()
{
 int m,n;
 int (*minus)(int,int) = subtraction;

 m = operation (7, 5, addition);
 n = operation (20, m, minus);
 cout <<n;
 return 0;
}
8

include <iostream>
using namespace std;

int main ()
{
 int firstvalue, secondvalue;
 int * mypointer;

 mypointer = &firstvalue;
 *mypointer = 10;
 mypointer = &secondvalue;
 *mypointer = 20;
 cout << "firstvalue is " << firstvalue << endl;
 cout << "secondvalue is " << secondvalue << endl;
 return 0;
}
```

```
firstvalue is 10
secondvalue is 20
```

```
// more pointers
#include <iostream>
using namespace std;

int main ()
{
  int firstvalue = 5, secondvalue = 15;
  int * p1, * p2;

  p1 = &firstvalue;  // p1 = address of firstvalue
  p2 = &secondvalue; // p2 = address of secondvalue
  *p1 = 10;          // value pointed by p1 = 10
  *p2 = *p1;         // value pointed by p2 = value pointed by p1
  p1 = p2;           // p1 = p2 (value of pointer is copied)
  *p1 = 20;          // value pointed by p1 = 20

  cout << "firstvalue is " << firstvalue << endl;
  cout << "secondvalue is " << secondvalue << endl;
  return 0;
}
```

```
firstvalue is 10
secondvalue is 20
```

```
// more pointers
#include <iostream>
using namespace std;

int main ()
{
  int numbers[5];
  int * p;
  p = numbers;  *p = 10;
  p++;  *p = 20;
```

```
 p = &numbers[2];  *p = 30;
 p = numbers + 3;  *p = 40;
 p = numbers;  *(p+4) = 50;
 for (int n=0; n<5; n++)
   cout << numbers[n] << ", ";
 return 0;
}
10, 20, 30, 40, 50,
```

كما اسلفنا فان المؤشر يمكن ان يشير الى موقع في الذاكرة وعليه فانه يمكن ان يكون مشيرا الى اي شيء مخزن فيها بحيث يمكن ان يشير الى مصفوفة او متجه او يشير الى اقتران او اي شيء اخر وفيما يلي بعض الامثلة والتي تبين كيفية استخدام المؤشرات للاشارة الى المتجهات:

```
int b[100];    // b is an array of 100 ints.
int* p;        // p is a pointer to an int.
p = b;         // Assigns the address of first element of b to p.
p = &b[0];     // Exactly the same assignment as above.



  p = b;  // Legal -- p is not a constant.
  b = p;  // ILLEGAL because b is a constant, altho the correct
type.

// Assume sizeof(int) is 4.
int b[100];  // b is an array of 100 ints.
int* p;      // p is a a pointer to an int.
p = b;       // Assigns address of first element of b. Ie, &b[0]
p = p + 1;   // Adds 4 to p (4 == 1 * sizeof(int)). Ie, &b[1]
```

```
int b[100];  // b is an array of 100 ints.
int* p;      // p is a a pointer to an int.
p = b;       // Assigns address of first element of b. Ie, &b[0]
*p = 14;     // Same as b[0] = 14
p = p + 1;   // Adds 4 to p (4 == 1 * sizeof(int)). Ie, &b[1]
*p = 22;     // Same as b[1] = 22;
```

```
int a[100];
...
int sum = 0;
for (int i=0; i<100; i++) {
  sum += a[i];
}
```

```
int a[100];
...
int sum = 0;
for (int* p=a; p<a+100; p++) {
  sum += *p;
}
```

وفيما يلي بعض الامثلة والتي توضح كيفية استخدام المؤشرات مع المتجهات او المصفوفات:

عند الاعلان عن المتجه فان اسم المتجه المستخدم في عملية الاعلان عن المتجه يستخدم كمؤشر وقيمته الابتدائية هي عنوان العنصر رقم صفر وعند زيادته فان تتم اضافة قيمة مساوية لعدد البايتات المخصصة للقيمة والتي ما تعتمد على نوع البيانات في المتجه.

مثال 1:

```
#include <iostream>
using namespace std;
const int Lengh = 3;
int main ()
{
  int testScore[Lengh] = {4, 7, 1};

  for (int I = 0; I < Lengh; i++)
  {
    cout << "The address of index " << I << " of the array is "<<
&testScore[i] << endl;
    cout << "The value at index " << I << " of the array is "<<
testScore[i] << endl;
  }
  return 0;
}
```

مثال 2:

```
#include <iostream>
using namespace std;
int main()
{
   int number[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

   cout << "List of Numbers";
   cout << "\nNumber 1:  " << number[0];
   cout << "\nNumber 2:  " << number[1];
   cout << "\nNumber 3:  " << number[2];
   cout << "\nNumber 4:  " << number[3];
   cout << "\nNumber 5:  " << number[4];
   cout << "\nNumber 6:  " << number[5];
   cout << "\nNumber 7:  " << number[6];
   cout << "\nNumber 8:  " << number[7];
   cout << "\nNumber 9:  " << number[8];
```

```
    cout << "\nNumber 10: " << number[9];
    cout << "\nNumber 11: " << number[10];
    cout << "\nNumber 12: " << number[11];

    return 0;
}
```

مثال 3:

```
#include <iostream>
using namespace std;

int main()
{
    int number[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

    cout << "\n Number    :  " << Number;
    cout << "\n&Number    :  " << &Number;
    cout << "\n&number[0]:  " << &number[0] << endl;

    return 0;
}
```

This would produce:

```
 Number    :  1245020

&Number    :  1245020

&number[0]:  1245020
```

مثال 4:

```
#include <iostream>

using namespace std;

int main()

{

   int number[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

   cout << "An integer occupies " << sizeof(int) << " bytes\n";

   cout << "\n Number:    " << Number;

   cout << "\n&number[0]: " << &number[0] << endl;

   cout << "\n Number+1:  " << Number+1;

   cout << "\n&Number:[1] " << &number[1] << endl;

   cout << "\n Number+2:  " << Number+2;

   cout << "\n&Number:[2] " << &number[2] << endl;

      return 0;

}
```

This would produce:

```
An integer occupies 4 bytes


 Number:    1245020
```

&number[0]: 1245020

Number+1: 1245024

&Number:[1] 1245024

Number+2: 1245028

&Number:[2] 1245028

مثال 5:

```
#include <iostream>
using namespace std;
int main()
{
   int number[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
   int *pNumbers = Number;

   cout << "Addresses";
   cout << "\n Number :  " << Number;
   cout << "\npNumbers:  " << pNumbers;
```

```
   return 0;
}
```

This would produce:

Addresses

Number : 1245020

pNumbers: 1245020

مثال 6:

```
#include <iostream>

using namespace std;

int main()
{
   int number[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
   int *pNumbers = Number;
   cout << "Values";
   cout << "\n  number[0]:  " << number[0];
```

```
   cout << "\n*pNumber   :  " << *pNumbers;
   return 0;
}
```

This would produce:

```
Values
 number[0]:  31
*pNumber   :  31
```

مثال 7:

```
#include <iostream>
using namespace std;
int main()
{
   int number[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
   int *pNumbers = Number;
   cout << "Addresses";
   cout << "\n Number :  " << Number;
   cout << "\npNumbers:  " << pNumbers;
   cout << "\n\nValues";
```

```
   cout << "\n Number [0]:  " << number[0];
   cout << "\npNumbers[0]:  " << pNumbers[0];
   cout << "\n Number [1]:  " << number[1];
   cout << "\npNumbers[1]:  " << pNumbers[1];


   return 0;
}
```

This would produce:

```
Addresses
 Number :  1245020
pNumbers:  1245020
Values
 Number [0]:  31
pNumbers[0]:  31
 Number [1]:  28
pNumbers[1]:  28
```

مثال 8:

```
#include <iostream>

using namespace std;

int main()
{
   int number[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
   int *pNumbers = Number;

   cout << "Addresses";
   cout << "\n Number   :  " << Number;
   cout << "\npNumbers  :  " << pNumbers;
   cout << "\n Number +1:  " << Number+1;
   cout << "\npNumbers+1:  " << pNumbers+1;
   cout << "\n Number +2:  " << Number+2;
   cout << "\npNumbers+2:  " << pNumbers+2;
```

```
   return 0;
}
```

This would produce:

```
Addresses
 Number   :  1245020
pNumbers  :  1245020
 Number +1:  1245024
pNumbers+1:  1245024
 Number +2:  1245028
pNumbers+2:  1245028
```

مثال 9:

```
#include <iostream>

using namespace std;

int main()
{
   int number[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
   int *pNumbers = Number;
```

```
cout << "Values - Using the Array";
cout << "\n number[0]:    " << number[0];
cout << "\n number[1]:    " << number[1];
cout << "\n number[2]:    " << number[2];
cout << "\n number[3]:    " << number[3];
cout << "\n number[4]:    " << number[4];

cout << "\n\nValues - Using the Pointer - No Parentheses";
cout << "\n*pNumbers:     " << *pNumbers;
cout << "\n*pNumbers+1:   " << *pNumbers+1;
cout << "\n*pNumbers+2:   " << *pNumbers+2;
cout << "\n*pNumbers+3:   " << *pNumbers+3;
cout << "\n*pNumbers+4:   " << *pNumbers+4;

cout << "\n\nValues - Using the Pointer - With Parentheses";
cout << "\n*pNumbers:     " << *pNumbers;
cout << "\n*(pNumbers+1): " << *(pNumbers+1);
cout << "\n*(pNumbers+2): " << *(pNumbers+2);
cout << "\n*(pNumbers+3): " << *(pNumbers+3);
```

```
    cout << "\n*(pNumbers+4): " << *(pNumbers+4);

    return 0;

}
```

This would produce:

```
Values - Using the Array

 number[0]:   31

 number[1]:   28

 number[2]:   31

 number[3]:   30

 number[4]:   31


Values - Using the Pointer - No Parentheses

*pNumbers:    31

*pNumbers+1:  32

*pNumbers+2:  33

*pNumbers+3:  34

*pNumbers+4:  35


Values - Using the Pointer - No Parentheses
```

```
*pNumbers:     31

*(pNumbers+1): 28

*(pNumbers+2): 31

*(pNumbers+3): 30

*(pNumbers+4): 31
```

مثال 10:

```
#include <iostream>

using namespace std;

int main()

{

   int number[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

   int *pNumbers = Number;

   int numberOfMembers = sizeof(Number) / sizeof(int);

   cout << "List of Numbers";

   for(int i = 0; i < NumberOfMembers; i++)

      cout << "\nNumber " << i + 1 << ": " << *(pNumbers+i);

   return 0;

}
```

تستخدم المؤشرات ايضا مع الاقترانات وقد يستخدم المؤشر للاشارة الى عنوان معلم من المعامل المرتبطة بالاقتران او قد يشير المؤشر الى الاقتران نفسه ولتوضيح هذا لنستعرض الامثلة التالية:

مثال 1:

البرنامج التالي يستخدم اقتران مرتبط بمعلم واحد حيث تمرر قيمة هذا المعلم من البرنامج الرئيسي لتحسب القيمة النهائية للسعر في الاقتران والتي تمرر منه الى البرنامج الرئيس:

هذا ويمكن ان يستخدم المؤشر كمعلم من معالم الاقتران باستخدام النجمة قبل اسم المعلم ولبيان هذا لناخذ نفس البرنامج في المثال 1 :

```
#include <iostream>

using namespace std;


double  CalculateNetPrice(double *disc);


int main()

{

   return 0;

}

double  CalculateNetPrice(double *discount)

{

   double origPrice;
```

```
    cout << "Please enter the original price: ";

    cin >> origPrice;

    return origPrice - (origPrice * *discount / 100);

}
```

عند استدعاء الاقتران اعلاه استخدم المرجع المسبوق باشارة والمنطقية كما مبين ادناه في عملية استدعاء الاقتران من البرنامج الرئيسي:

```
int main()

{

    double finalPrice;

    double discount = 20;

    finalPrice = CalculateNetPrice(&discount);


    cout << "\nAfter applying a 20% discount";

    cout << "\nFinal Price = " << finalPrice << "\n";


    return 0;

}
```

An example of running the program is:

Please enter the original price: 100

After applying a 20% discount

Final Price = 80

مثال 2:

تمرير المؤشرات كمعالم Passing Pointers as Arguments

Create a new project named Fire Insurance2
Create a C++ source file named Main.cpp
Change the Main.cpp file as follows:

```
 #include <iostream>

using namespace std;


double GetAnnualPremium();

double GetCoverage();

double GetPolicy();

double CalculatePremium(double Rt, double Cvr, double Plc);


int main()

{

   double Rate, Coverage, Policy, Premium;


   cout << "Fire Insurance - Customer Processing\n";

   Rate     = GetAnnualPremium();
```

```
   Coverage = GetCoverage();

   Policy  = GetPolicy();

   Premium = CalculatePremium(Rate, Coverage, Policy);

   cout << "\n********************************";

   cout << "\nFire Insurance - Customer Quote";

   cout << "\n________________________________";

   cout << "\nAnnual Premium: $" << Rate;

   cout << "\nCoverage:      $" << Coverage;

   cout << "\nPolicy:        $" << Policy;

   cout << "\nPremium:       $" << Premium;

   cout << "\n********************************\n";

   return 0;
}

double GetAnnualPremium()
{
   double AnlPrem;

   cout << "Enter the annual premium: $";
```

```
    cin >> AnlPrem;

    return AnlPrem;

}

double GetCoverage()

{

    double Cover;


    cout << "Enter the coverage: $";

    cin >> Cover;

    return Cover;

}

double GetPolicy()

{

    double Plc;


    cout << "Enter the policy amount: $";

    cin >> Plc;

    return Plc;

}
```

```
double CalculatePremium(double Rate, double Cover, double Pol)
{
   double Prem;
   int Unit;

   Unit = Pol / Cover;
   Prem = Rate * Unit;
   return Prem;
}
```

Test the program. Here is an example:

```
 Fire Insurance - Customer Processing

Enter the annual premium: $0.55
Enter the coverage: $92
Enter the policy amount: $45000

********************************

Fire Insurance - Customer Quote
________________________________

Annual Premium: $0.55
Coverage:      $92
```

Policy: $45000

Premium: $268.95

****

مثال 2:

لناخذ البرنامج الرئيسي التالي والذي يستخدم اقترانا بدون ان يمرر قيمة المعلم منه الى البرنامج الرئيسي:

```
#include <iostream>

using namespace std;

void GetTheOriginalPrice(double OrigPrice);

int main()
{
   double OriginalPrice = 0;

   cout << "First in main() --";
   cout << "\nOriginal Price =  $" << OriginalPrice << endl;

   GetTheOriginalPrice(OriginalPrice);
```

```
   cout << "\nBack in main() --";
   cout << "\nOriginal Price =  $" << OriginalPrice << endl;

   return 0;
}

void GetTheOriginalPrice(double OrigPrice)
{
   cout << "\nNow we are in the GetTheOriginalPrice() function";
   cout << "\nPlease enter the original price: ";
   cin >> OrigPrice;

   cout << "\nIn the GetTheOriginalPrice() function";
   cout << "\nOriginal Price = $" << OrigPrice << endl;
}
```

Here is an example of running the program:

```
First in main() --
Original Price =  $0

Now we are in the GetTheOriginalPrice() function
Please enter the original price: 100
```

In the GetTheOriginalPrice() function

Original Price = $100

Back in main() --

Original Price = $0

ولـو اسـتخدمنا موقـع المعلـم كمعلـم فـان البرنـامج الرئيسـي والاقـتران سيصـلان الى هـذا الموقـع او بمعنـى اخـر سـتتم عمليـة التمريـر مـن الاقـتران الى البرنـامج الرئيسي اي ان اي تغـيير علـى القيمـة المخزنـة في الموقـع سـتكون متاحـة للبرنامج الرئيسي:

```
#include <iostream>

using namespace std;


void GetTheOriginalPrice(double *OrigPrice);


int main()

{

   double OriginalPrice = 0;


   cout << "First in main() --";

   cout << "\nOriginal Price =  $" << OriginalPrice << endl;
```

```
    GetTheOriginalPrice(&OriginalPrice);


    cout << "\nBack in main() --";
    cout << "\nOriginal Price =  $" << OriginalPrice << endl;


    return 0;
}


void GetTheOriginalPrice(double *OrigPrice)
{
    cout << "\nNow we are in the GetTheOriginalPrice() function";
    cout << "\nPlease enter the original price: ";
    cin >> *OrigPrice;


    cout << "\nIn the GetTheOriginalPrice() function";
    cout << "\nOriginal Price = $" << *OrigPrice << endl;
}
```

Here is an example of executing this program:

First in main() --

Original Price = $0

Now we are in the GetTheOriginalPrice() function

Please enter the original price: 100

In the GetTheOriginalPrice() function

Original Price = $100

Back in main() --

Original Price = $100

مثال 3:

لمعالجة المتغيرات باستخدام المؤشرات والمراجع اجري التعديلات التالية على البرنامج السابق:

```
#include <iostream>
using namespace std;


void GetAnnualPremium(double *Prem);
void GetCoverage(double *Cvr);
```

```
void GetPolicy(double *Plc);

double CalculatePremium(double *Rt, double *Cvr, double *Plc);


int main()

{

    double Rate, Coverage, Policy, Premium;


    cout << "Fire Insurance - Customer Processing\n";

    GetAnnualPremium(&Rate);

    GetCoverage(&Coverage);

    GetPolicy(&Policy);


    Premium = CalculatePremium(&Rate, &Coverage, &Policy);


    cout << "\n********************************";

    cout << "\nFire Insurance - Customer Quote";

    cout << "\n________________________________";

    cout << "\nAnnual Premium: $" << Rate;

    cout << "\nCoverage:       $" << Coverage;
```

```
    cout << "\nPolicy:        $" << Policy;
    cout << "\nPremium:       $" << Premium;
    cout << "\n*******************************\n";

    return 0;
}

void GetAnnualPremium(double *AnlPrem)
{
    cout << "Enter the annual premium: $";
    cin >> *AnlPrem;
}

void GetCoverage(double *Cover)
{
    cout << "Enter the coverage: $";
    cin >> *Cover;
}

void GetPolicy(double *Plc)
```

```
{
   cout << "Enter the policy amount: $";
   cin >> *Plc;
}

double CalculatePremium(double *Rate, double *Cover, double
*Pol)
{
   double Prem;
   int Unit;

   Unit = *Pol / *Cover;
   Prem = *Rate * Unit;
   return Prem;
}
```

Test the application. Here is an example:

```
 Fire Insurance - Customer Processing

Enter the annual premium: $0.74

Enter the coverage: $120
```

Enter the policy amount: $60000

****

Fire Insurance - Customer Quote

Annual Premium: $0.74

Coverage: $120

Policy: $60000

Premium: $370

****

عند استقبال الاقتران المؤشر كمعلم فمن المفترض ان لا يغير الاقتران قيمة المؤشر او العنوان وعليه بامكانك تمرير المؤشر كثابت وعليه تحجب عملية تعديل العنوان نهائيا:

```
#include <iostream>

using namespace std;

double CalculateNetPrice(const double *Disc);

int main()

{
```

```
    double FinalPrice;

    double Discount = 20;



    FinalPrice = CalculateNetPrice(&Discount);


    cout << "\nAfter applying a 20% discount";

    cout << "\nFinal Price = " << FinalPrice << "\n";


    return 0;

}

double  CalculateNetPrice(const double *Discount)

{

    double OrigPrice;

    cout << "Please enter the original price: ";

    cin >> OrigPrice;

    return OrigPrice - (OrigPrice * *Discount / 100);

}
```

مثال 4:

تمرير المؤشرات كثوابت:

لناخذ البرنامج في المثال السابق ونستخدم المؤشرات كثوابت:

```
#include <iostream>
using namespace std;

void GetAnnualPremium(double *Prem);
void GetCoverage(double *Cvr);
void GetPolicy(double *Plc);
double CalculatePremium( const double *Rt, const double *Cvr,
                         const double *Plc );

int main()
{
   double Rate, Coverage, Policy, Premium;
   cout << "Fire Insurance - Customer Processing\n";
   GetAnnualPremium(&Rate);
   GetCoverage(&Coverage);
   GetPolicy(&Policy);
```

```
   Premium = CalculatePremium(&Rate, &Coverage, &Policy);

   cout << "\n*******************************";

   cout << "\nFire Insurance - Customer Quote";

   cout << "\n______________________________";

   cout << "\nAnnual Premium: $" << Rate;

   cout << "\nCoverage:      $" << Coverage;

   cout << "\nPolicy:        $" << Policy;

   cout << "\nPremium:       $" << Premium;

   cout << "\n*******************************\n";


   return 0;

}

void GetAnnualPremium(double *AnlPrem)

{

   cout << "Enter the annual premium: $";

   cin >> *AnlPrem;

}

void GetCoverage(double *Cover)
```

```
{
   cout << "Enter the coverage: $";
   cin >> *Cover;
}

void GetPolicy(double *Plc)
{
   cout << "Enter the policy amount: $";
   cin >> *Plc;
}

double CalculatePremium (const double *Rate, const double
*Cover,
                         const double *Pol)
{
   double Prem;
   int Unit;
   Unit = *Pol / *Cover;
   Prem = *Rate * Unit;
   return Prem;
}
```

اشـرنا في الامثلـة السـابقة الى كيفيـة التعامـل مـع المتجهـات او المصفوفات احاديـة البعـد باسـتخدام المؤشـرات وبـنفس الآليـة يمكـن التعامـل مـع المصـفوفات متعددة الابعاد باستخدام المؤشرات والامثلة التالية تبين كيفية استخدام المؤشـرات مع المصفوفات متعددة الابعاد:

مثال 1:

البرنامج التالي يتعامل مع مصفوفة ثنائية البعد ويطبع مقوع العنصر وقيمته:

```
#include <iostream>

using namespace std;

int main()
{
   int number[2][6] = { { 31, 28, 31, 30, 31, 30 },
                        { 31, 31, 30, 31, 30, 31 } };
   cout << "List of Numbers";
   for(int i = 0; i < 2; i++)
```

```
    for(int j = 0; j < 6; j++)
        cout << "\nNumber [" << i << "][" << j << "]: " <<
number[i][j];

  return 0;
}
```

بالامكان الان استخدام المؤشرات للتعامل مع المصفوفة وكما هو مبين في البرنامج ادناه:

```
#include <iostream>
using namespace std;
int main()
{
  int number[2][6] = { { 31, 28, 31, 30, 31, 30 },
                       { 31, 31, 30, 31, 30, 31 } };
  int *pNumbers[2];

  *pNumbers = number[0];
  (*pNumbers)[0]    = number[0][0];
  (*pNumbers)[1]    = number[0][1];
  (*pNumbers)[2]    = number[0][2];
```

```
(*pNumbers)[3]    = number[0][3];

(*pNumbers)[4]    = number[0][4];

(*pNumbers)[5]    = number[0][5];


*(pNumbers+1) = number[1];

(*(pNumbers+1))[0] = number[1][0];

(*(pNumbers+1))[1] = number[1][1];

(*(pNumbers+1))[2] = number[1][2];

(*(pNumbers+1))[3] = number[1][3];

(*(pNumbers+1))[4] = number[1][4];

(*(pNumbers+1))[5] = number[1][5];


cout << "List of Numbers";

cout << "\n(*pNumbers)[0]    = " << (*pNumbers)[0];

cout << "\n(*pNumbers)[1]    = " << (*pNumbers)[1];

cout << "\n(*pNumbers)[2]    = " << (*pNumbers)[2];

cout << "\n(*pNumbers)[3]    = " << (*pNumbers)[3];

cout << "\n(*pNumbers)[4]    = " << (*pNumbers)[4];

cout << "\n(*pNumbers)[5]    = " << (*pNumbers)[5] << endl;
```

```
   cout << "\n(*(pNumbers+1))[0] = " << (*(pNumbers+1))[0];

   cout << "\n(*(pNumbers+1))[1] = " << (*(pNumbers+1))[1];

   cout << "\n(*(pNumbers+1))[2] = " << (*(pNumbers+1))[2];

   cout << "\n(*(pNumbers+1))[3] = " << (*(pNumbers+1))[3];

   cout << "\n(*(pNumbers+1))[4] = " << (*(pNumbers+1))[4];

   cout << "\n(*(pNumbers+1))[5] = " << (*(pNumbers+1))[5] <<
endl;


   return 0;

}
```

This would produce:

```
List of Numbers

(*pNumbers)[0]    = 31

(*pNumbers)[1]    = 28

(*pNumbers)[2]    = 31

(*pNumbers)[3]    = 30

(*pNumbers)[4]    = 31

(*pNumbers)[5]    = 30
```

```
(*(pNumbers+1))[0] = 31

(*(pNumbers+1))[1] = 31

(*(pNumbers+1))[2] = 30

(*(pNumbers+1))[3] = 31

(*(pNumbers+1))[4] = 30

(*(pNumbers+1))[5] = 31
```

عند استخدام المؤشرات مع المصفوفات فانه يمكن حجز وتخصيص مجموعة من المواقع ديناميكيا وذلك لتخزين قيم عناصر المصفوفة في المواقع التي تم حجزها والمثال التالي يبين كيفية تنفيذ عملية الحجز الديناميكي للمصفوفة:

```
double *Distance = new double[12];

unsigned int *pRanges = new unsigned int[120];

float *Prices = new float[44];
```

بعد عملية الحجز هذه فاننا نستطيع الوصول الى المواقع لوضع البيانات فيها كما يلي:

```
int *pNumbers = new int[12];


pNumbers[0] = 31;

pNumbers[1] = 29;

pNumbers[2] = 31;

pNumbers[3] = 30;
```

بامكانك ايضا الوصول الى عناوين العناصر المخزنة في الذاكرة كما يلي:

```
int *pNumbers = new int[12];


*(pNumbers+4) = 31;

*(pNumbers+5) = 30;

*(pNumbers+6) = 31;

*(pNumbers+7) = 31;
```

وهذه التعليمات مكافئة للتعليمات السابقة حيث استخدمنا هنا العناوين بدلا من استخدام الفهرس. والبرنامج التالي يبين كيفية تنفيذ عملية الحجز الديناميكي للمصفوفة:

```
#include <iostream>

using namespace std;


int main()

{

   int *pNumbers = new int[12];


     pNumbers[0]  = 31;

     pNumbers[1]  = 29;
```

```
pNumbers[2]  = 31;
pNumbers[3]  = 30;
*(pNumbers+4)  = 31;
*(pNumbers+5)  = 30;
*(pNumbers+6)  = 31;
*(pNumbers+7)  = 31;
*(pNumbers+8)  = 30;
*(pNumbers+9)  = 31;
pNumbers[10] = 30;
pNumbers[11] = 31;

cout << "List of numbers";
cout << "\nNumber 1:  " << *pNumbers;
cout << "\nNumber 2:  " << *(pNumbers+1);
cout << "\nNumber 3:  " << *(pNumbers+2);
cout << "\nNumber 4:  " << *(pNumbers+3);
cout << "\nNumber 5:  " << pNumbers[4];
cout << "\nNumber 6:  " << pNumbers[5];
cout << "\nNumber 7:  " << pNumbers[6];
```

```
    cout << "\nNumber 8:  " << pNumbers[7];

    cout << "\nNumber 9:  " << *(pNumbers+8);

    cout << "\nNumber 10: " << *(pNumbers+9);

    cout << "\nNumber 11: " << pNumbers[10];

    cout << "\nNumber 12: " << pNumbers[11];


    return 0;

}
```

This would produce:

```
List of numbers

Number 1:  31

Number 2:  29

Number 3:  31

Number 4:  30

Number 5:  31

Number 6:  30

Number 7:  31

Number 8:  31

Number 9:  30
```

Number 10: 31

Number 11: 30

Number 12: 31

بعد الحجز الديناميكي في الذاكرة يمكن إلغاء عملية الحجز وذلك باستخدام تعليمة الحذف كما يلي:

```
#include <iostream>


using namespace std;

int main()

{

  int number[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

  int *pNumbers = Number;

  int numberOfMembers = sizeof(Number) / sizeof(int);


  cout << "List of Numbers";

  for(int i = 0; i < NumberOfMembers; i++)

     cout << "\nNumber " << i + 1 << ": " << *(pNumbers+i);


  delete [] pNumbers;
```

```
   return 0;
}
```

تنفذ عادة عملية الألغاء بعد عملية الحجز الديناميكي والمثال التالي يبين كيفية تنفيذ هذه العملية:

```
#include <iostream>
using namespace std;
int main()
{
   const int Size = 12;
   int *pNumbers = new int[Size];

     pNumbers[0]  = 31;
     pNumbers[1]  = 28;
     pNumbers[2]  = 31;
     pNumbers[3]  = 30;
   *(pNumbers+4)  = 31;
   *(pNumbers+5)  = 30;
   *(pNumbers+6)  = 31;
   *(pNumbers+7)  = 31;
```

```
    *(pNumbers+8)  = 30;
    *(pNumbers+9)  = 31;
     pNumbers[10] = 30;
     pNumbers[11] = 31;


    cout << "List of numbers";
    for(int i = 0; i < Size; i++)
        cout << "\nNumber " << i + 1 << ":  " << *(pNumbers+i);


    delete [] pNumbers;
    pNumbers = NULL;
    return 0;
}
```

تنفذ عملية الحجز الديناميكي للمصفوفات متعددة الابعاد بنفس الآلية المستخدمة مع المصفوفات احادية البعد وكما هو مبين في البرنامج التالي:

```
#include <iostream>

using namespace std;
int main()
```

```
{
  int *pNumbers[2];


  *pNumbers = new int[0];
  (*pNumbers)[0]    = 31;
  (*pNumbers)[1]    = 29;
  (*pNumbers)[2]    = 31;
  (*pNumbers)[3]    = 30;
  (*pNumbers)[4]    = 31;
  (*pNumbers)[5]    = 30;


  *(pNumbers+1) = new int[1];
  (*(pNumbers+1))[0] = 31;
  (*(pNumbers+1))[1] = 31;
  (*(pNumbers+1))[2] = 30;
  (*(pNumbers+1))[3] = 31;
  (*(pNumbers+1))[4] = 30;
  (*(pNumbers+1))[5] = 31;
  cout << "List of Numbers";
```

```
cout << "\n(*pNumbers)[0]    = " << (*pNumbers)[0];

cout << "\n(*pNumbers)[1]    = " << (*pNumbers)[1];

cout << "\n(*pNumbers)[2]    = " << (*pNumbers)[2];

cout << "\n(*pNumbers)[3]    = " << (*pNumbers)[3];

cout << "\n(*pNumbers)[4]    = " << (*pNumbers)[4];

cout << "\n(*pNumbers)[5]    = " << (*pNumbers)[5] << endl;



cout << "\n(*(pNumbers+1))[0] = " << (*(pNumbers+1))[0];

cout << "\n(*(pNumbers+1))[1] = " << (*(pNumbers+1))[1];

cout << "\n(*(pNumbers+1))[2] = " << (*(pNumbers+1))[2];

cout << "\n(*(pNumbers+1))[3] = " << (*(pNumbers+1))[3];

cout << "\n(*(pNumbers+1))[4] = " << (*(pNumbers+1))[4];

cout << "\n(*(pNumbers+1))[5] = " << (*(pNumbers+1))[5] <<
endl;



delete [] *pNumbers;

delete [] *(pNumbers+1);



return 0;
```

}

This would produce;

List of Numbers

(*pNumbers)[0] = 31

(*pNumbers)[1] = 29

(*pNumbers)[2] = 31

(*pNumbers)[3] = 30

(*pNumbers)[4] = 31

(*pNumbers)[5] = 30

(*(pNumbers+1))[0] = 31

(*(pNumbers+1))[1] = 31

(*(pNumbers+1))[2] = 30

(*(pNumbers+1))[3] = 31

(*(pNumbers+1))[4] = 30

(*(pNumbers+1))[5] = 31

يمكن ان تستخدم المتجهات او المصفوفات كمعالم مرتبطة بالاقتران وفي هذه الحالة يستطيع البرنامج الرئيسي والاقتران الوصول الى عناصر المصفوفة

باستخدام الاسم كمرجع او عنوان او استخدام المؤشر والامثلة التالية تبين كيفية استخدام المصفوفات كمعالم في الاقترانات:

- Single Dimensional Arrays and Functions

```
#include <iostream>

using namespace std;

int  SumOfNumbers(int Nbr[], int Size)

{

   int Sum = 0;

   for(int i = 0; i < Size; i++)

      Sum += Nbr[i];

  return Sum;

}
int main()

{

   int number[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

   int numberOfMembers = sizeof(Number) / sizeof(int);


   int Value = SumOfNumbers(number, numberOfMembers);
   cout << "Sum of numbers: " << Value;


   return 0;
```

```
}
```

This would produce:

Sum of numbers: 365
The above program can also be written as follows:

```
#include <iostream>

using namespace std;

int  SumOfNumbers(int *nbr, int size)

{

   int sum = 0;

   for(int i = 0; i < size; i++)

      sum += nbr[i];


   return Sum;

}


int main()

{

   int number[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

   int *pNumbers = number;

   int numberOfMembers = sizeof(number) / sizeof(int);
```

```
   int Value = SumOfNumbers(pNumbers, numberOfMembers);

   cout << "Sum of numbers: " << Value;

      return 0;

}
```

This would produce the same result.

-Multi-Dimensional Arrays and Functions

```
 #include <iostream>
using namespace std;

void  DisplayNumbers(int *Nbr[]);

int main()
{
   int number[2][6] = { { 31, 28, 31, 30, 31, 30 },
                   { 31, 31, 30, 31, 30, 31 } };

   int *pNumbers[2];

   *pNumbers = number[0];
   (*pNumbers)[0]     = number[0][0];
   (*pNumbers)[1]     = number[0][1];
   (*pNumbers)[2]     = number[0][2];
   (*pNumbers)[3]     = number[0][3];
   (*pNumbers)[4]     = number[0][4];
   (*pNumbers)[5]     = number[0][5];

   *(pNumbers+1) = number[1];
   (*(pNumbers+1))[0] = number[1][0];
   (*(pNumbers+1))[1] = number[1][1];
   (*(pNumbers+1))[2] = number[1][2];
```

```
   (*(pNumbers+1))[3] = number[1][3];
   (*(pNumbers+1))[4] = number[1][4];
   (*(pNumbers+1))[5] = number[1][5];

   cout << "List of Numbers";
   DisplayNumbers(pNumbers);

   return 0;
}

void  DisplayNumbers(int *nbr[])
{
   cout << "\n(*pNumbers)[0]    = " << (*nbr)[0];
   cout << "\n(*pNumbers)[1]    = " << (*nbr)[1];
   cout << "\n(*pNumbers)[2]    = " << (*nbr)[2];
   cout << "\n(*pNumbers)[3]    = " << (*nbr)[3];
   cout << "\n(*pNumbers)[4]    = " << (*nbr)[4];
   cout << "\n(*pNumbers)[5]    = " << (*nbr)[5] << endl;

   cout << "\n(*(pNumbers+1))[0] = " << (*(nbr+1))[0];
   cout << "\n(*(pNumbers+1))[1] = " << (*(nbr+1))[1];
   cout << "\n(*(pNumbers+1))[2] = " << (*(nbr+1))[2];
   cout << "\n(*(pNumbers+1))[3] = " << (*(nbr+1))[3];
   cout << "\n(*(pNumbers+1))[4] = " << (*(nbr+1))[4];
   cout << "\n(*(pNumbers+1))[5] = " << (*(nbr+1))[5] << endl;
}
```

```
#include <iostream>
using namespace std;

void  DisplayNumbers(int *Nbr[], int r, int c);

int main()
{
   int number[2][6] = {{ 31, 28, 31, 30, 31, 30},
                       { 31, 31, 30, 31, 30, 31} };
```

```
    int *pNumbers[2];

    *pNumbers = number[0];

    for(int i = 0; i < 6; i++)
        (*pNumbers)[i] = number[0][i];

    *(pNumbers+1) = number[1];

    for(int i = 0; i < 6; i++)
        (*(pNumbers+1))[i] = number[1][i];

    cout << "List of Numbers";
    DisplayNumbers(pNumbers, 2, 6);

    return 0;
}


void  DisplayNumbers(int *nbr[], int rows, int columns)

{

    for(int i = 0; i < rows; i++)

        for(int j = 0; j < columns; j++)

            cout << "\nNumber[" << i << "][" << j << "]: " <<
(*(nbr+i))[j];
}
```

Here is an example of executing this program:

List of Numbers

Number[0][0]: 31

Number[0][1]: 28

Number[0][2]: 31

Number[0][3]: 30

Number[0][4]: 31

Number[0][5]: 30

Number[1][0]: 31

Number[1][1]: 31

Number[1][2]: 30

Number[1][3]: 31

Number[1][4]: 30

Number[1][5]: 31

يمكن استخدام المؤشر للاشارة الى الاقتران والمثال التالي يبين كيفية استخدام المؤشر للاشارة الى الاقتران:

```
// pointer to functions
#include <iostream>
using namespace std;

int addition (int a, int b)
{ return (a+b); }

int subtraction (int a, int b)
{ return (a-b); }
```

```
int operation (int x, int y, int (*functocall)(int,int))
{
 int g;
 g = (*functocall)(x,y);
 return (g);
}

int main ()
{
 int m,n;
 int (*minus)(int,int) = subtraction;

 m = operation (7, 5, addition);
 n = operation (20, m, minus);
 cout <<n;
 return 0;
}
8
```

الوحدة الثانية

2

الأصناف

CLASSES

الاصناف

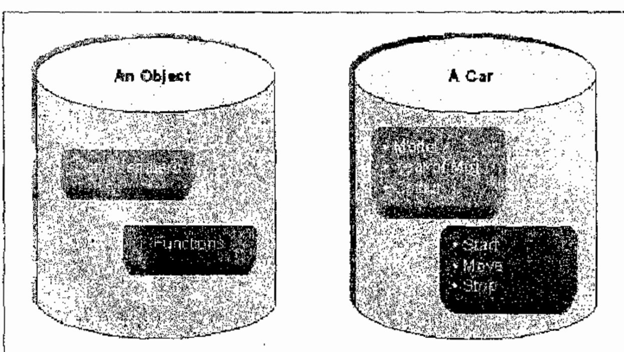
CLASSES

يعـرف الصـنف علـى انـه نـوع مـن انـواع البيانـات المعلـن عنهـا مـن قبـل المستخدم والتي يمكن استخدما للإعلان عن اهداف معينة في البرنامج.

يعتبر الصنف من اهم الاشياء التي تزودنا بها لغة سي بلس بلس والتي تستخدم للاعلان عن اهداف متعددة وعليه تسمى البرمجة المستخدمة للاصناف والاهداف برمجة الكيانات الموجهه.

يتضمن الصنف مجموعة من الاعضاء هي:

- عضو البيانات.
- عضور التعليمات او الاقترانات او ما يسمى طريقة المعالجة وكما هو مبين في الشكل التالي:



قد يحتوي الصنف الواحد على عضو البيانات وعضو طريقة المعالجة او يمكن ان يحتوي فقط على عضو البيانات او عضو المعالجة فقط والامر منوط بالمستخدم والوظائف المطلوبة من الاهداف المعلن عنها باستخدام الصنف.

الاعلان عن الصنف:

يتم الاعلان عن الصنف في سي بلس بلس باستخدام الصيغة التالية:

تستخدم الكلمة المحجوزة صنف متبوعة باسم الصنف ومن ثم جسم الصنف والذي يتضمن اعضاء الصنف كما يلي:

```
class class_name {
 access_specifier_1:
   member1;
 access_specifier_2:
   member2;
 ...
 } object_names;
```

حيث يمكن ان تكون الاعضاء بيانات او طرق معالجة او كلاهما معا.

والامثلة التالية تبين كيفية الاعلان عن الصنف:

```
#include <iostream>
using namespace std;
class z1
{
   public:

   int a,b;
}singleton;

int main()
{
   //singleton s;  #cannot define a new instant like this anymore.
   singleton.a=5;
   cout<<"a="<<singleton.a;
      return 0;
}
```

المثال اعلاه يعرف صنف يحتوي على عضوي بيانات وهذا الصنف يمكن استخدامه لتعريف هدف واحد فقط لاحظ ان عملية تعريف الهدف تمت مباشرة في نهايــة عمليــة الاعــلان عــن الصــنف. ولاتاحــة الفرصــة لتعريــف اكثــر مــن هــدف باستخدام نفس الصنف يتم الاعلان عن الاهداف في البرنامج الرئيسي.

لناخذ الصنف السابق ونستخدمه للاعلان عن هدفين وكما هو مبين في المثال التالي:

```
#include <iostream>
using namespace std;

class z1
{
  public:

  int a;
};

int main()
{
 Z1 obj1,obj2; //declare object 1 and 2
Obj1.a=5;
Obj1.b=9;
Obj2.a=8;
Obj2,b=12;
Cout<<"\n object 1 elements:"<<obj1.a<<"   "<<obj1.b;
Cout<<"\n object 2 elements:"<<obj2.a<<"   "<<obj2.b;


  return 0;
}
```

لاحظ ان اعضاء الاهداف المعرفة باستخدام الصنف في المثال السابق تضمنت بيانات فقط ويمكن للصنف ان يتضمن ايضا عضو طريقة المعالجة والذي قد يتكون من اقتران او اكثر وكما هو مبين في المثال التالي:

```
#include <iostream>
using namespace std;

class z2
{
   public:
   void print()
   {
      Cout<<"\nhello\n";
   }
   int a,b;
};

int main()
{
 Z2 obj1,obj2; //declare object 1 and 2
Obj1..print;;
Obj2.print;
 return 0;
}
```

يتم تعريف الاقترانات اما داخل الصنف او خارجه والمثال التالي يبين كيفية تعريف الاقتران داخل الصنف:

```
class x
{
public:
     int add()          // inline member function add
     {return a+b+c;};
private:
     int a,b,c;
};
```

هذا ويمكن ان يتم الاعلان عن الاقتران خارج الصنف وفي هذه الحالة لا بد من ربط اسم الاقتران بالصنف باستخدام عامل الربط والممثل باربعة نقاط وكما هو مبين في المثال التالي:

```
// example: one class, two objects
#include <iostream>
using namespace std;

class CRectangle {
   int x, y;
  public:
   void set_values (int,int);
   int area () {return (x*y);}
};

void CRectangle::set_values (int a, int b) {
  x = a;
  y = b;
}

int main () {
  CRectangle rect, rectb;
  rect.set_values (3,4);
  rectb.set_values (5,6);
  cout << "rect area: " << rect.area() << endl;
  cout << "rectb area: " << rectb.area() << endl;
  return 0;
}
rect area: 12
rectb area: 30
```

يشير واصف المعالجة الى ميزة فريدة في الصنف تستخدم لغايات حماية اعضاء الصنف من عمليات الاستخدام والتي يمكن ان تكون احد الاشكال التالية:

- الواصف العام وفي هذه الحالة يجوز استخدام العضو من اي موقع في البرنامج.
- الواصف الخاص وفي هذه الحالة يستخدم العضو من قبل الصنف من قبل الاصناف الصديقة.
- واصف الحماية وفي هذه الحالة لا يجوز استخدام العضو الا من قبل الصنف او الاصناف المشتقة منه او من قبل الاصناف الصديقة الامثلة التالية تبين كيفية التعامل مع هذه الواصفات:

```
#include <iostream>
using namespace std;
//Private: Class members declared as private can be used
only //by member functions and friends (classes or
functions) of the //class
// keyword_ private.cpp
class BaseClass {
public:
  // privMem accessible from member function
  int pubFunc() { return privMem; }
private:
  void privMem;
};

class DerivedClass: public BaseClass {
public:
  void usePrivate( int i )
    { privMem = i; }   // C2248: privMem not accessible
                 // from derived class
};

class DerivedClass2: private BaseClass {
public:
```

```
  // pubFunc() accessible from derived class
  int usePublic() { return pubFunc(); }
};

int main() {
  BaseClass aBase;
  DerivedClass aDerived;
  DerivedClass2 aDerived2;
  aBase.privMem = 1;     // C2248: privMem not accessible
  aDerived.privMem = 1;  // C2248: privMem not accessible
                  //    in derived class
  aDerived2.pubFunc();   // C2247: pubFunc() is private in
                  //    derived class
}
```

//Public: Class members declared as public can be used by //any function.

```
// keyword_public.cpp
class BaseClass {
public:
  int pubFunc() { return 0; }
};

class DerivedClass: public BaseClass {};

int main() {
  BaseClass aBase;
  DerivedClass aDerived;
  aBase.pubFunc();       // pubFunc() is accessible
                  //    from any function
  aDerived.pubFunc();    // pubFunc() is still public in
                  //    derived class
}
```

//Protected: Class members declared as protected can be //used by member functions and friends (classes or functions) //of the class. Additionally, they can be used by classes //derived from the class.

```
// keyword_protected.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
class X {
public:
  void setProtMemb( int i ) { m_protMemb = i; }
  void Display() { cout << m_protMemb << endl; }
protected:
  int  m_protMemb;
  void Protfunc() { cout << "\nAccess allowed\n"; }
} x;

class Y: public X {
public:
  void useProtfunc() { Protfunc(); }
} y;

int main() {
  // x.m_protMemb;        error, m_protMemb is protected
  x.setProtMemb( 0 );   // OK, uses public access function
  x.Display();
  y.setProtMemb( 5 );   // OK, uses public access function
  y.Display();
  // x.Protfunc();        error, Protfunc() is protected
  y.useProtfunc();      // OK, uses public access function
                        // in derived class
}
```

والان لنعرف صنف يستخدم فقط عضو البيانات وكما هو مبين في البرنامج التالي:

```
#include <iostream>

using namespace std;
class Hotel {

int  roomcount;
float occrate;
};

int main () {
Hotel manor;
Hotel beechfield;
manor.roomcount = 6;
beechfield.roomcount = 18;
manor.occrate = 0.85;
beechfield.occrate = 0.35;

int totrooms = manor.roomcount + beechfield.roomcount;

cout << "Total rooms listed: " << totrooms << "\n" ;

return 0;
}
```

لاحظ انه لم يتم تحديد واصف الوصول الى البيانات او واصف الاستخدام الامر الذي قد يوقع المستخدم في الأخطاء وعلية لا بد من تحديد واصف الاستخدام كما يلي:

```
#include <iostream>

using namespace std;
class Hotel {
```

```
public:
int  roomcount;
float occrate;
};

int main () {
Hotel manor;
Hotel beechfield;
manor.roomcount = 6;
beechfield.roomcount = 18;
manor.occrate = 0.85;
beechfield.occrate = 0.35;

int totrooms = manor.roomcount + beechfield.roomcount;

cout << "Total rooms listed: " << totrooms << "\n" ;

return 0;
}
```

لناخذ البرنامج التالي:

```
// DateClass.cc
// Program to demonstrate the definition of a simple class
// and member functions

#include <iostream>
using namespace std;


// Declaration of Date class
class Date {

public:
  Date(int, int, int);
  void set(int, int, int);
  void print();
```

```
private:
 int year;
 int month;
 int day;
};


int main()
{
  // Declare today to be object of class Date
  // Values are automatically intialised by calling constructor
//function
  Date today(1,9,1999);

  cout << "This program was written on ";
  today.print();

  cout << "This program was modified on ";
  today.set(5,10,1999);
  today.print();

  return 0;
}

// Date constructor function definition
Date::Date(int d, int m, int y)
{
 if(d>0 && d<31) day = d;
 if(m>0 && m<13) month = m;
 if(y>0) year =y;
}

// Date member function definitions
void Date::set(int d, int m, int y)
{
 if(d>0 && d<31) day = d;
 if(m>0 && m<13) month = m;
 if(y>0) year =y;
```

```
}

void Date::print()
{
 cout << day << "-" << month << "-" << year << endl;
}
```

استخدم البرنامج السابق صنف اتوى على عضو بيانات خاص مؤلف من 3 متغيرات صحيحة وعلى عضو عام لطريقة المعالجة تالف من 3 اقترانات تم تعريفها خارج الصنف وقد تم استخدام اقترانين هما اقتران الطباعة واقتران اعطاء القيم للمتغيرات اما الاقتران الثالث فقد تمت تسميته باسم الصنف ولكن لم يستعى هذا الاقتران.

يسمى الاقتران الذي يعرف باستخدام اس الصنف الغضو المهيئ او الباني constructor بحيث يتم تنفيذه اوتوماتيكيا ومجرد التعامل مع الهدف المعلن عنه باستخدام الصنف حيث يعمل هذا المهيئ على اعطاء القيم الابتدائية المنصوص عليها في تعليمات المهيئ. وسوف نعود الى المهيئ في هذه الوحدة.

لاحظ انه يمكن تعريف الاقتران داخل الصنف وكثال على هذا انظر الى الصنف التالي:

```
class Date
    {
      public:
    int m_nMonth;
    int m_nDay;
    int m_nYear;

    void SetDate(int nMonth, int nDay, int nYear)
               {
```

```
10        m_nMonth = nMonth;
11        m_nDay = nDay;
12        m_nYear = nYear;
13    }
14 };
```

تم الاعلان عن الاقتران مباشرة داخل الصنف (الاسطر 8 الى 13) ويمكن استخدام هذا الاقتران من البرنامج الرئيسي بنفس الطريقة التي تعلمناها سابقا وكما يلي:

```
1                    Date cToday;
2 cToday.SetDate(10, 14, 2020); // call SetDate() on cToday
```

حيث استخدم السطر الاول للاعلان عن الهدف في البرنامج الرئيسي اما السطر الثاني فاستخدم لاستدعاء الاقتران الخاص بالهدف لتمرير القيم المشار اليها الى متغيرات الهدف.

وفيما يلي برنامج اخر يستخدم صنفا عرفت فيه اقتراناته داخل الصنف:

```
01 #include <iostream>
02    class Employee
03    {
04 public:
05    char m_strName[25];
06          int m_nID;
07    double m_dWage;
08
09                // Set the employee information
10    void SetInfo(char *strName, int nID, double dWage)
11                {
```

```
        strncpy(m_strName, strName, 25);
            m_nID = nID;
        m_dWage = dWage;
    }

    // Print employee information to the screen
                        void Print()
                  {
        using namespace std;
         cout << "Name: " << m_strName << "  Id: " <<
            m_nID << "  Wage: $" << m_dWage << endl;
    }
};

int main()
                {
    // Declare two employees
            Employee cAlex;
    cAlex.SetInfo("Alex", 1, 25.00);

    Employee cJoe;
    cJoe.SetInfo("Joe", 2, 22.25);

    // Print out the employee information
                  cAlex.Print();
    cJoe.Print();

    return 0;
     }
```

This produces the output:

Name: Alex Id: 1 Wage: $25
Name: Joe Id: 2 Wage: $22.25

أشرنا سابقا الى ضرورة تحديد واصفات الاستخدام وضرورة فهم الية التعامل مع البيانات واقترانات المستخدمة في الصنف. واذا لم تتم عملية تحديد واصف الاستخدام فان الواصف المرجعي سيكون الواصف الخاص وعلى سبيل المثال لناخذ البرنامج التالي:

```
01 class Date
02     {
03     int m_nMonth;
04      int m_nDay;
05     int m_nYear;
06         };
07
08 int main()
09         {
10     Date cDate;
11     cDate.m_nMonth = 10;
12      cDate.m_nDay = 14;
13     cDate.m_nYear = 2020;
14
15     return 0;
16      }
```

في الاسطر 3 الى 5 تم الاعلان عن عضو البيانات والمؤلف من 3 متغيرات ولم يحدد لهذا العضو واصف الاستخدام وعليه فانه يعتبر خاصا وعند استخدام هذه المتغيرات في البرنامج الرئيسي فان المترجم سوف يعلن عن خطأ لابد من

تصحيحه حتى تستطيع تنفيذ هذا البرنامج وعليه وحتى تصبح الاسطر 11 – 13 صحيحة وبدون اخطاء لا بد من اجراء التعديل التالي على البرنامج:

```
01 class Date
02    {
03           public:
04    int m_nMonth; // public
05        int m_nDay; //
           public
06    int m_nYear; // public
07 };
08
09 int main()
10    {
11                          Date cDate;
12    cDate.m_nMonth = 10; // okay because m_nMonth is public
13      cDate.m_nDay = 14;  // okay because m_nDay is public
14    cDate.m_nYear = 2020;  // okay because m_nYear is public
15
16    return 0;
17 }
```

لاحظ الاضافة في السطر الثالث وفي هذه الحالة تستطيع ترجمة البرنامج وتنفيذه.

وفيما يلي برنامج يستخدم الواصفات الثلاثة والتي اشرنا اليها سابقا:

```
01 class Access
02    {
03            int m_nA; // private by default
```

```
int GetA() { return m_nA; } // private by default

private:
        int m_nB; // private
int GetB() { return m_nB; } // private

protected:
        int m_nC; // protected
int GetC() { return m_nC; } // protected

public:
        int m_nD; // public
int GetD() { return m_nD; } // public

};

int main()
    {
 Access cAccess;
     cAccess.m_nD = 5; // okay because m_nD is public
    std::cout << cAccess.GetD(); // okay because GetD() is
                                //public

 cAccess.m_nA = 2; // WRONG because m_nA is private
  std::cout << cAccess.GetB(); // WRONG because GetB() is
                               //private

 return 0;
  }
```

لاحظ التعليقات في الاسطر 23 – 27.

اقترانات المعالجة وكبسلة البيانات:

Access functions and ecapsulation:

اقتران المعالجة ما هو الا اقتران عام وقصير ومؤلف من بعض التعليمات والتي يؤدي تنفيذها الى ارجاع قيم اعضاء البيانات الخاصة والمعرفة في الصنف. لنأخذ البرنامج التالي:

```
1 class String
2     {
3                         private:
4     char *m_chString; // a dynamically allocated string
5     int m_nLength; // the length of m_chString
6
7                    public:
8     int GetLength() { return m_nLength; }
9 }
```

الاقتران المعرف في السطر الثامن ما هو الا اقتران معالجة يعمل على ارجاع قيمة متغير خاص معرف في الصنف ولا نستطيع الوصول اليه من البرنامج الرئيسي.

لاحظ كيفية التعامل مع اقترانات المعالجة في المثال التالي:

```
class Date
    {
     private:
    int m_nMonth;
         int
    m_nDay;
```

```
int m_nYear;

public:
                    // Getters
int GetMonth() { return m_nMonth; }
 int GetDay() { return m_nDay; }
int GetYear() { return m_nYear; }

// Setters
void SetMonth(int nMonth) { m_nMonth = nMonth; }
     void SetDay(int nDay) { m_nDay = nDay; }
void SetYear(int nYear) { m_nYear = nYear; }
                      };
```

مما سبق نستطيع طرح السؤال الهام التالي:

ما هو الداعي الى استخدام اعضاء البيانات الخاص؟ ولنستخدم دائما المتغيرات العامة.

الاجابة على هذا السؤال توضح مفهوم كبسلة البيانات وهو موضوع مهم جدا عند التعامل مع الاهداف والبرمجة الموجهة.

لنأخذ المثال التالي:

```
class Change
  {
   public:
int m_nValue;
};

```

```
int main()
{
Change cChange;
cChange.m_nValue = 5;
std::cout << cChange.m_nValue << std::endl;
}
```

ماذا لو اردنا تغيير اسم المتغيرm_nValue؟

في هذه الحالة فان عملية التغيير سيصاحبها احداث خلل في البرنامج ولحل هذه المشكلة لا بد من اللجوء الى عملية كبسلة البيانات وباستخدام اقترانات المعالجة التي اشرنا اليها سابقا في هذا البند.

لناخذ المثال التالي:

```
class Change
{
private:
int m_nValue;

public:
void SetValue(int nValue) { m_nValue = nValue; }
int GetValue() { return m_nValue; }
};

int main()
{
Change cChange;
cChange.SetValue(5);
std::cout << cChange.GetValue() << std::endl;
}
```

والان اذا قررنا تغيير اسم المتغير m_nValue ما علينا فقط هو احداث بعض التغيير في الاقترانات SetValue and GetValue() لتنفيذ التغيير المطلوب في الاسم.

العضو المهيئ او الباني:

Constructor:

العضو المهيئ ما هو الا اقتران خاص من الاقترانات المرتبطة بالصنف والذي ينفذ اوتوماتيكيا عند بدء عملية التعامل مع الهدف المعلن عنه باستخدام الصنف.

وعند التعامل مع عضو التهيئة لابد من الاخذ بما يلي:

- اسم هذا العضو يجب ان يكون مطابقا لاسم الصنف.
- لا يحتوي المهيئ على اي نوع من البيانات الراجعة (no return type).

يسمى المهيئ الذي لا يرتبط بمعالم بالمهيئ المرجعي ويعمل هذا المهيئ على اعطاء القيم الابتدائية للمتغيرات فور الاعلان عن الهدف ومباشرة بعد حجز الذاكرة للهدف المعلن عنه باستخدام الصنف ولبيان هذا لناخذ المثال التالي:

```
class Fraction
    {
      private:
int m_nNumerator;
int m_nDenominator;

            public:
Fraction() // default constructor
            {
```

```
10        m_nNumerator = 0;
11        m_nDenominator = 1;
12            }
13
14    int GetNumerator() { return m_nNumerator; }
15        int GetDenominator() { return m_nDenominator; }
16            double GetFraction() { return
      static_cast<double>(m_nNumerator) / m_nDenominator; }
17 };
```

تضمن هذا المثال استخدام مهيئ مرجعي في الاسطر 8 – 12 والذي يعمل على تهيئه الهدف باعطاء المتغيرات القيم الابتدائية المشار اليها في المهيئ.

لاحظ انه اذا استخدمنا الجمل التالية في البرنامج الرئيسي فان تنفيذها سيولد المخرجات المشار اليها:

```
Fraction cDefault; // calls Fraction() constructor
std::cout << cDefault.GetNumerator() << "/" <<
    cDefault.GetDenominator() << std::endl;
```

produces the output:

0/1

قد يشتمل المهيئ على معالم لناخذ الان المثال التالي:

```
#include <cassert>
class Fraction
{
private:
    int m_nNumerator;
    int m_nDenominator;

```

```
public:
  Fraction() // default constructor
  {
      m_nNumerator = 0;
      m_nDenominator = 1;
  }

  // Constructor with parameters
  Fraction(int nNumerator, int nDenominator=1)
  {
     assert(nDenominator != 0);
     m_nNumerator = nNumerator;
     m_nDenominator = nDenominator;
  }

  int GetNumerator() { return m_nNumerator; }
  int GetDenominator() { return m_nDenominator; }
  double GetFraction() { return static_cast<double>(m_nNumerator) / m_nDenominator; }
};
```

اشتمل هذا البرنامج على عضوي تهيئة الاول مرجعي بدون معالم والثاني بمعالم . ينفذ المهيئ الأول مباشرة بعد الاعلان عن الهدف اما المهيئ الثاني فيمكن استخدامه متى شئنا وباسم المهيئ متبوعا باسم تختاره كما تشاء. لاحظ الاستدعاء التالي ونتيجة الطباعة:

```
Fraction cFiveThirds(5, 3); // calls Fraction(int, int) constructor
```

لاحظ هنا ان المهيئ المرجعي يمكن اعتباره فائضا ويمكن الاستغناء عنه لهذا المثال ليصبح البرنامج كما يلي:

```
#include <cassert>
class Fraction
{
private:
    int m_nNumerator;
    int m_nDenominator;

public:
    // Default constructor
    Fraction(int nNumerator=0, int nDenominator=1)
    {
        assert(nDenominator != 0);
        m_nNumerator = nNumerator;
        m_nDenominator = nDenominator;
    }

    int GetNumerator() { return m_nNumerator; }
    int GetDenominator() { return m_nDenominator; }
    double GetFraction() { return
static_cast<double>(m_nNumerator) / m_nDenominator; }
};
```

ويمكن استدعاء هذا المهيئ كما يلي:

```
Fraction cDefault; // will call Fraction(0, 1)
Fraction cSix(6); // will call Fraction(6, 1)
Fraction cFiveThirds(5,3); // will call Fraction(5,3)
```

لكن ماذا لو لم يتم الاعلان عن المهيئ المرجعي في الصنف؟

في هذه الحالة سيتم استحداث الهدف وحجز الذاكرة له دون معرفة ما هو مخزن في المواقع التي تم تخصيصها للمتغيرات لننظر الى البرنامج التالي:

```
class Date
{
private:
    int m_nMonth;
    int m_nDay;
    int m_nYear;
};

int main()
{
    Date cDate;
    // cDate's member variables now contain garbage
    // Who knows what date we'll get?

    return 0;
}
```

وعليه وللتخلص من هذه المشكلة نستخدم المهيئ وكما هو مبين في البرنامج التالي:

```
class Date
{
private:
    int m_nMonth;
```

```
    int m_nDay;
    int m_nYear;

public:
    Date(int nMonth=1, int nDay=1, int nYear=1970)
    {
        m_nMonth = nMonth;
        m_nDay = nDay;
        m_nYear = nYear;
    }
};

int main()
{
    Date cDate; // cDate is initialized to Jan 1st, 1970 instead of
//garbage

    Date cToday(3, 9, 2011); // cToday is initialized to March
//9th, 2007

    return 0;
}
```

كما يتعامل الصنف مع عضو البناء والتهيئة فانه يتعمل ايضا مع عنصر الهدم والذي يتم تفعيله بعد انهاء معالجة الهدف والغاء الذاكرة المخصصة لهذا الهدف.

عند التعامل مع عضو الهدم لا بد من الاخذ بالأمور التالية:

- يعرف عضو الهدم باستخدام اسم الصنف مسبوقا بالاشارة ~
- لا يرتبط عضو الهدم باية معالم.
- لا توجد قيم راجعة لعضو الهدم.

والمثال التالي يبين كيفية الاعلان عن عضو الهدم وكيفية استخدامه في البرنامج:

```
class MyString
{
private:
    char *m_pchString;
    int m_nLength;

public:
    MyString(const char *pchString="")
    {
        // Find the length of the string
        // Plus one character for a terminator
        m_nLength = strlen(pchString) + 1;

        // Allocate a buffer equal to this length
        m_pchString = new char[m_nLength];

        // Copy the parameter into our internal buffer
        strncpy(m_pchString, pchString, m_nLength);

        // Make sure the string is terminated
        m_pchString[m_nLength-1] = '\0';
```

```
    }

    ~MyString() // destructor
    {
        // We need to deallocate our buffer
        delete[] m_pchString;

        // Set m_pchString to null just in case
        m_pchString = 0;
    }

    char* GetString() { return m_pchString; }
    int GetLength() { return m_nLength; }
};
```

والان لنبين كيفية استخدام هذا العضو:

```
int main()
{
    MyString cMyName("ODAI");
    std::cout << "My name is: " << cMyName.GetString() << std::endl;
    return 0;
} // cMyName destructor called here!
```

This program produces the result:

My name is: ODAI

لعضو البناء او التهيئة وعضو الهدم توقيت محدد فالاول ينفذ بعد الاعلان عن الهدف باستخدام الصنف والثاني ينفذ بعد الانتهاء من معالجة الهدف. لناخذ

المثال التالي والذي يوضح هذه الامور حيث استخدمنا جملة الطباعة داخل كل من عضو البناء وعضو الهدم:

```
class Simple
{
private:
    int m_nID;

public:
    Simple(int nID)
    {
        std::cout << "Constructing Simple " << nID<< std::endl;
        m_nID = nID;
    }

    ~Simple()
    {
        std::cout << "Destructing Simple" << m_nID << std::endl;
    }

    int GetID() { return m_nID; }
};

int main()
{
    // Allocate a Simple on the stack
    Simple cSimple(1);
    std::cout << cSimple.GetID() << std::endl;

```

```
// Allocate a Simple dynamically
Simple *pSimple = new   Simple(2);


std::cout << pSimple->GetID() << std::endl;
delete pSimple;

return 0;
} // cSimple goes out of scope here
```

This program produces the following result:

```
Constructing Simple 1
1
Constructing Simple 2
2
Destructing Simple 2
Destructing Simple 1
```

لاحظ مخرجات البرنامج التالي:

```
#include <iostream>
using namespace std;

class myclass {
 int a;
public:
 myclass();            // constructor
 ~myclass();            // destructor
 void show();
};

myclass::myclass()
{
 cout << "In constructor\n";
 a = 10;
}
```

```
myclass::~myclass()
{
 cout << "Destructing...\n";
}

void myclass::show()
{
 cout << a << endl;
}

int main()
{
 myclass ob;

 ob.show();

 return 0;
}
```

```
In constructor
10
Destructing...
```

لاحظ مخرجات البرنامج التالي:

```
#include <iostream>
using namespace std;

class myclass {
public:
 int who;
 myclass(int id);
 ~myclass();
};

myclass::myclass(int id)
{
```

```
 stack(); // constructor
 ~stack(); // destructor
 void push(int i);
 int pop();
};

// constructor
stack::stack(){
 topOfStack = 0;
 cout << "Stack Initialized\n";
}

// destructor
stack::~stack(){
 cout << "Stack Destroyed\n";
}

void stack::push(int i){
 if( topOfStack == SIZE ) {
  cout << "Stack is full.\n";
  return;
 }
 stck[ topOfStack ] = i;
 topOfStack++;
}

int stack::pop() {
 if( topOfStack == 0 ) {
  cout << "Stack underflow.\n";
  return 0;
 }
 topOfStack--;
 return stck[ topOfStack ];
}

int main()
{
 stack a, b;
```

```
 a.push(1);
 b.push(2);

 a.push(3);
 b.push(4);

 cout << a.pop() << " ";
 cout << a.pop() << " ";
 cout << b.pop() << " ";
 cout << b.pop() << endl;

 return 0;
}
```

```
Stack Initialized
Stack Initialized
3 1 4 2
Stack Destroyed
Stack Destroyed
```

نفذ البرنامج التالي ولاحظ النتيجة:

```
#include <iostream>
using namespace std;

class prompt {
 int count;
public:
 prompt(char *s) {
   cout << s; cin >> count;
 };
 ~prompt();
};

prompt::~prompt() {
 int i, j;
```

```
  for(i = 0; i <count; i++) {
    cout << '\a';
    for(j=0; j<32000; j++)
      ; // delay
  }
}

int main()
{
  prompt ob("Enter a number: ");

  return 0;
}
```

```
Enter a number: 1
```

The hidden "this" pointer:المؤشر هذا

من احد الاسئلة المهمة والتي قد يطرحها متعلم البرمجة هو: كيف يتم استدعاء الاقتران العضو ولاي هدف يتبع هذا الاقتران؟ كيف يحدد سي بلس بلس الاقتران وتبعية الاقتران؟

للاجابة على هذا السؤال تستخدم سي بلس بلس مؤشرا مخفيا يسمى المؤشر "هذا".

لناخذ الصنف التالي:

```
class Simple
{
private:
    int m_nID;

public:
```

```
    Simple(int nID)
    {
        SetID(nID);
    }

    void SetID(int nID) { m_nID = nID; }
    int GetID() { return m_nID; }
};
```

يمكن استخدام هذا الصنف في البرنامج كما يلي:

```
int main()
{
    Simple cSimple(1);
    cSimple.SetID(2);
    std::cout << cSimple.GetID() << std::endl;
}
```

وبما ان سي بلس بلس تترجم امر الاستدعاء فانها تترجم ايضا الاقتران نفسه فتعليمة الاستدعاء التالية:

```
void SetID(int nID) { m_nID = nID; }
```

تصبح كما يلي:

```
void SetID(Simple* const this, int nID)
 { this->m_nID = nID; }
```

لناخذ الصنف التالي:

```
class Calc
{
private:
    int m_nValue;

public:
    Calc() { m_nValue = 0; }

    void Add(int nValue) { m_nValue += nValue; }
    void Sub(int nValue) { m_nValue -= nValue; }
    void Mult(int nValue) { m_nValue *= nValue; }

    int GetValue() { return m_nValue; }
};
```

واذا اردت زيادة 5 وطرح 3 والضرب ب 4 فانه بامكانك تنفيذ التالي:

```
Calc cCalc;
cCalc.Add(5);
cCalc.Sub(3)


cCalc.Mult(4);
```

وباستخدام مؤشر "هذه" يمكن اعادة كتابة الصنف السابق كما يلي:

```
class Calc
{
private:
    int m_nValue;

public:
    Calc() { m_nValue = 0; }

    Calc& Add(int nValue) { m_nValue += nValue; return *this; }
    Calc& Sub(int nValue) { m_nValue -= nValue; return *this; }
    Calc& Mult(int nValue) { m_nValue *= nValue; return *this; }

    int GetValue() { return m_nValue; }
};
```

اما عملية الاستدعاء فيمكن ان تنفذ كما يلي:

```
Calc cCalc;
cCalc.Add(5).Sub(3).Mult(4);
```

اشرنا سابقا الى العضو المهيئ وكنا قد استخدمناه كعضو عام؟ لكن ماذا عن منع عملية التهيئة من خارج الصنف؟

في هذه الحالة لا بد من تعريف عضو التهيئة كعضو خاص يمكن ان يستخدم فقط من داخل اقترانات الصنف والمثال التالي يبين كيفية استخدام عضو التهيئة الخاص:

```
class Book
{
private:
    int m_nPages;

    // This constructor can only be used by Book's members
   Book() // private default constructor
    {
        m_nPages = 0;
    }

public:
    // This constructor can be used by anybody
    Book(int nPages) // public non-default //constructor
    {
       m_nPages = nPages;
    }
};

int main()
{
    Book cMyBook; // fails because default constructor Book() is private
    Book cMyOtherBook(242); // okay because Book(int) is public

    return 0;
}
```

في بعض الاحيان قد تشترك اعضاء التهيئة في استخدام بعض الاقترانات كما هو موضح في المثال التالي:

```
class Foo
{
public:
    Foo()
    {
        // code to do A
    }

    Foo(int nValue)
    {
        // code to do A
        // code to do B
    }
};
```

ولحل هذا التكرار يمكن اعادة كتابة الصنف السابق كما يلي:

```
class Foo
{
public:
    Foo()
    {
        DoA();
    }

    Foo(int nValue)
    {
        DoA();
        // code to do B
```

```
    }

    void DoA()
    {
        // code to do A
    }
};
```

والمثال التالي يبين كيفية التعامل مع الاقترانات المستخدمة من قبل اكثر من عضو تهيئة:

```
class Foo
{
public:
    Foo()
    {
        Init();
    }

    Foo(int nValue)
    {
        Init();
        // do something with nValue
    }

    void Init()
    {
        // code to init Foo
    }
};
```

لاحظ ان الصنف قد يحتوي على اكثر من عضوء تهيئة لناخذ البرنامج التالي:

```
// overloading class constructors
#include <iostream>
using namespace std;

class CRectangle {
   int width, height;
  public:
   CRectangle ();
   CRectangle (int,int);
   int area (void) {return (width*height);}
};

CRectangle::CRectangle () {
  width = 5;
  height = 5;
}

CRectangle::CRectangle (int a, int b) {
  width = a;
  height = b;
}

int main () {
  CRectangle rect (3,4);
  CRectangle rectb;
  cout << "rect area: " << rect.area() << endl;
  cout << "rectb area: " << rectb.area() << endl;
  return 0;
}
```

لاحظ هنا ان الهدف الاول استخدم عضو التهيئة المرجعي الاول اما الهدف الثاني فاستخدم عضو التهيئة الثاني وعليه تكون نتيجة تنفيذ هذا البرنامج كما يلي:

```
rect area: 12
rectb area: 25
```

3

الوحدة الثالثة

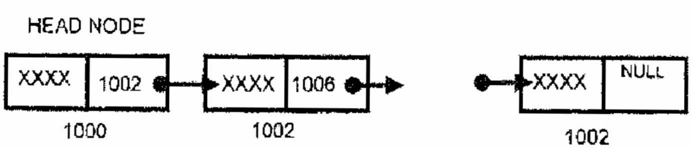
الصنف والمؤشرات

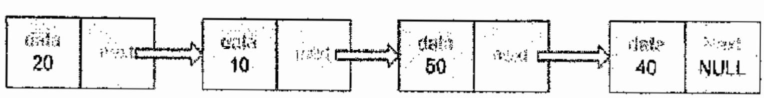
الصنف والمؤشرات

اشرنا سابقا ان الصنف يتضمن مجموعة من اعضاء البيانات والإجراءات او الاقترانات وان عضو البيانات يمكن ان يحتوي على اي نوع من البيانات بما فيها المؤشرات.

وقبل الحديث عن المؤشرات المرتبطة بالصنف لنتذكر بعض الأمور الهامة والمتعلقة بالمؤشرات.

يتألف المتجه من مجموعة من العناصر المخزنة في الذاكرة بحيث يخزن كل عنصر من العناصر في موقع أو أكثر وعليه فإننا لو تعاملنا مع العنصر كصنف كل عنصر فيه مؤلف من القيمة ومؤشر يشير الى موقع العنصر التالي فاننا نحصل على قائمة متصلة وكما هو مبين في الشكل التالي:





Linked list

وعليه فإن العنصر في القائمة يمكن ان يعرف كما يلي:

```
Linkedlist Node {
   data // The value or data stored in the node
   next // A reference to the next node, null for last node
}
```

وقبل الحديث عن الصنف المخصص للتعامل مع عنصر القائمة لنسترجع بعض المعلومات عن المؤشرات:

يتم التعامل مع اسم المتغير كمؤشر انظر الاعلان التالي:

int a = 50 // initialize variable a



ويمكن اعطاؤه قيمة كما يلي:

a = 100 // new initialization

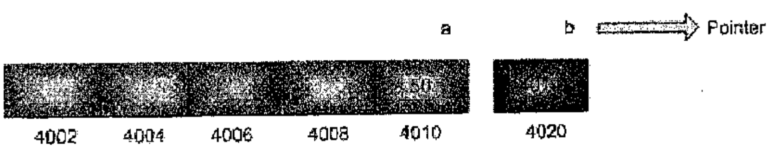
اما عملية الاعلان عن مؤشر فتتم كما يلي:

int *b; // declare pointer b

ويمكن وضع قيمة عنوان المتغير السابق في المؤشر السابق كما يلي:

b = &a;

the unary operator & gives the address of an object



ويمكن تغيير قيمة المتغير الان باستخدام المؤشر كما يلي:

*b = 100; // change the value of 'a' using pointer 'b'

```
cout<<a; // show the output of 'a'
```

هذا ويمكن استخدام المؤشر للاشارة الى مؤشر كما يلي:

```
int **c;  //declare a pointer to a pointer
 c = &b;  //transfer the address of 'b' to 'c'
```

ويمكن تغيير قيمة المتغير كما يلي:

```
**c = 200;
// change the value of 'a' using pointer to a pointer 'c'
 cout<<a; // show the output of a
```

والان ادرس البرنامج التالي لتلاحظ الية التعامل مع المؤشرات:

```
#include<iostream>
using namespace std;

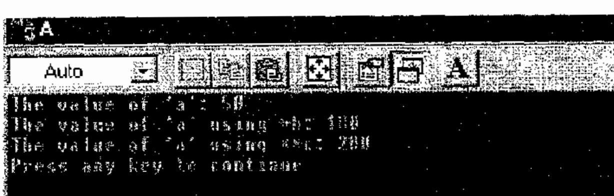

int main()
{
    int a = 50;      // initialize integer variable a
    cout<<"The value of 'a': "<<a<<endl;
// show the output of a


    int * b;        // declare an integer pointer b
    b = &a;
 // transfer the address of 'a' to pointer 'b'
```

```
    *b = 100;
// change the value of 'a' using pointer 'b'
    cout<<"The value of 'a' using *b: "<<a<<endl;
// show the output of a


    int **c;        // declare an integer pointer to pointer 'c'
    c = &b;
// transfer the address of 'b' to pointer to pointer 'c'
    **c = 200;
// change the value of 'a' using pointer to pointer 'c'
    cout<<"The value of 'a' using **c: "<<a<<endl;
// show the output of a
    return 0;
}
```

بعد تنفيذ هذا البرنامج فاننا سنحصل على النتائج التالية:



والآن لنأخذ البرنامج التالي:

```
#include<iostream>
using namespace std;
```

```
int main()
{
    int a = 50;
// initialize integer variable a
    cout<<"Value of 'a' = "<<a<<endl;
      // show the output of a
    cout<<"Memory address of 'a': "<<&a<<endl;
 // show the address of a
    cout<<endl;
    int * b;
 // declare an integer pointer b
    b = &a;
      // transfer the address of 'a' to pointer 'b'
    cout<<"Value of Pointer 'b': "<<*b<<endl;
// show the output of *b
    cout<<"Content of Pointer 'b': "<<b<<endl;
 // show the content of *b
    cout<<"Memory address of Pointer 'b': "<<&b<<endl; // show
the address of *b
    cout<<endl;


    int **c;
          // declare an integer pointer to a pointer
```

```
    c = &b;

      // transfer the address of 'b' to 'c'

    cout<<"Value of Pointer 'c': "<<**c<<endl;

// show the output of **c

    cout<<"Content of Pointer 'c': "<<c<<endl;

 // show the content of **c

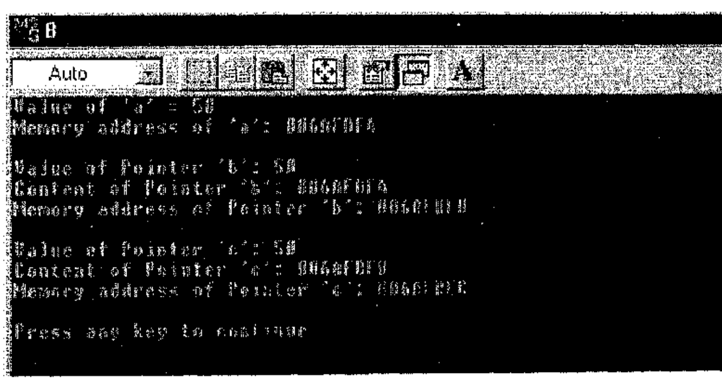
    cout<<"Memory address of Pointer 'c': "<<&c<<endl; // show
the address of **c

    cout<<endl;

    return 0;

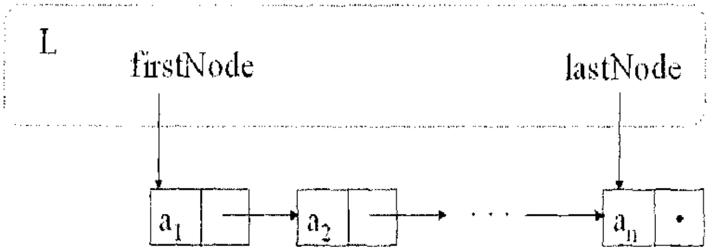
}
```

سيعطي هذا البرنامج المخرجات التالية:

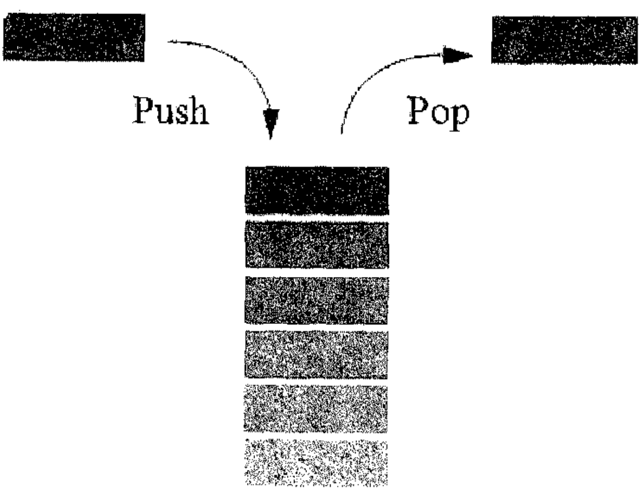


للاصناف تطبقات مهمة في معالجة تراكيب البيانات المختلفة ومن هذه التراكيب:

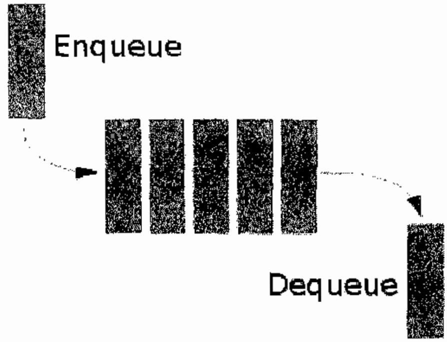
- القيمة المتصلة وهي مجموعة من العناصر بحيث يتضمن كل عنصر فيها البيانات ومؤشر يشير الى العنصر التالي وكما هو موضح في الشكل التالي:



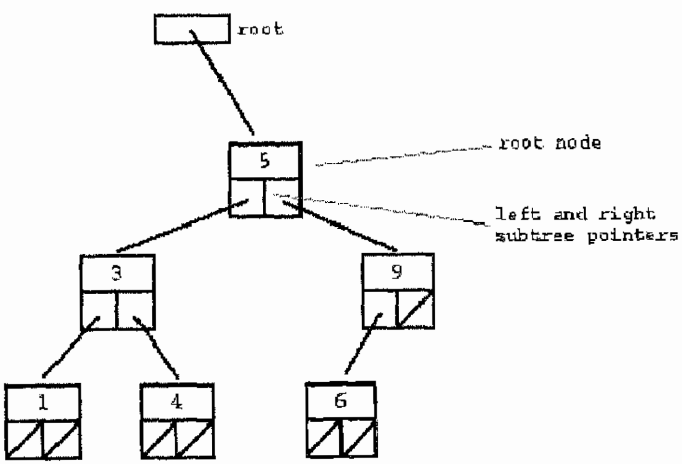
- الحزمة وهي مجموعة من العناصر تقبل الاضافة والحذف من طرف واحد ألا وهو نهاية الحزمة وكما هو مبين في الشكل التالي:



- الطابور وهو هيكل بيانات مؤلف من مجموعة من العناصر تقبل الاضافة في نقطة النهاية والحذف من نقطة البداية وكما هو مبين في الشكل التالي:



- الهيكل الشجري الثنائي ويمتلك كل عنصر فيه مؤشرين واحد للاشارة الى الطرف الايسر والاخر للاشارة الى الطرف الايمن وكما هو مبين في الشكل التالي:



والان لننظر كيف نعالج القائمة المتصلة باستخدام الصنف والمؤشرات:

اولا نعلن عن صنف القائمة والذي سيستخدم للاعلان عن الاهداف الخاصة بالقائمة والذي يمكن ان يكون كما يلي:

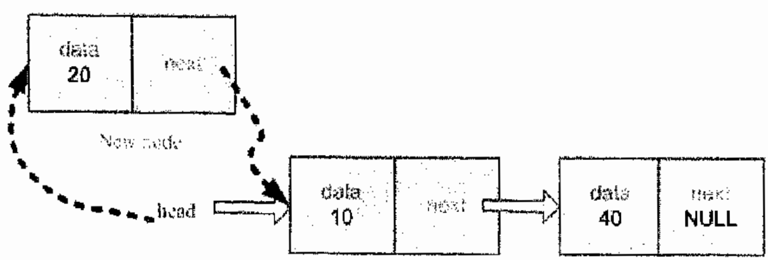
```
class node {
    int data;
        // will store information
    node *next;
      // the reference to the next node
};
```

لاستحداث عنصر من عناصر القائمة ننفذ ما يلي:

```
node *head = NULL;          //empty linked list
```

```
node *temp;           //create a temporary node
temp = new node;
 //allocate space for node
```

بعد ذلك نفذ التعليمات التالية لاعطاء القيم وتغيير مؤشر القائمة:



Linked list

```
temp->data = info;          // store data(first field)

temp->next=head;

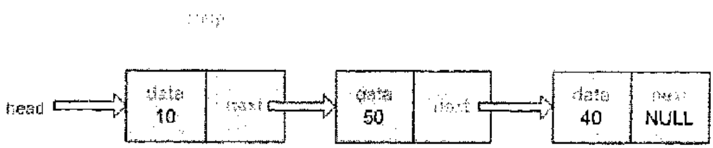
 // store the address of the pointer head(second field)

head = temp;

        // transfer the address of 'temp' to 'head'
```

لاسترجاع عناصر القائمة نفذ التعليمات التالية:

```
while( temp1!=NULL )
{
 cout<< temp1->data<<" ";
// show the data in the linked list
 temp1 = temp1->next;
// tranfer the address of 'temp->next' to 'temp'
}
```

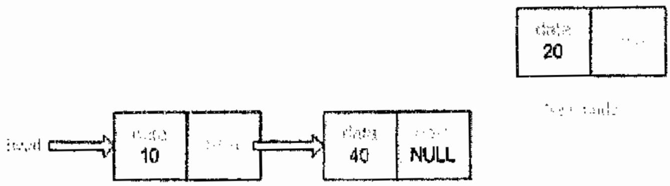


Linked list

للإضافة في نهاية القائمة نفذ التعليمات التالية:

```
node *temp1;                    // create a temporary node
temp1=new node;
 // allocate space for node
temp1 = head;
            // transfer the address of 'head' to 'temp1'
while(temp1->next!=NULL)
 // go to the last node
    temp1 = temp1->next;
//tranfer the address of 'temp1->next' to 'temp1'
```

والان استحدث عقدة او عنصر مؤقت كما يلي:



Linked list

```
node *temp;

                    // create a temporary node

temp =new node;

 // allocate space for node

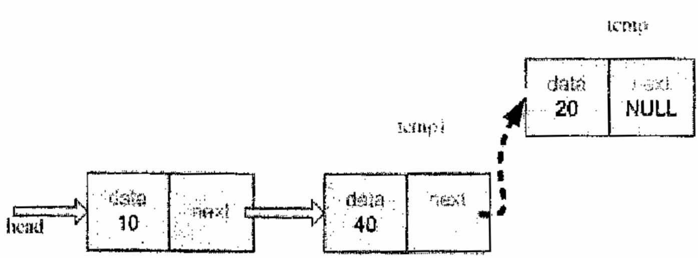
temp->data = info;                // store data(first field)

temp->next = NULL;

  // second field will be null(last node)

temp1->next = temp;

    // 'temp' node will be the last node
```



Linked list

ولتنفيذ عملية الادخال بعد عدد محدد من العناصر نفذ ما يلي:

```
cout<<"ENTER THE NODE NUMBER:";
cin>>node_number;              // take the node number from user

node *temp1;                   // create a temporary node
temp1 = new node;; // allocate space for node
temp1 = head;

for( int i = 1 ; i < node_number ; i++ )
{
    temp1 = temp1->next;          // go to the next node

    if( temp1 == NULL )
    {
        cout<<node_number<<" node is not exist"<< endl;
        break;
    }
}
```

والان استحدث عقدة مؤقتة:

```
node *temp;                    // create a temporary node

temp = new node;

 // allocate space for node

temp->data = info;

             // store data(first field)
```

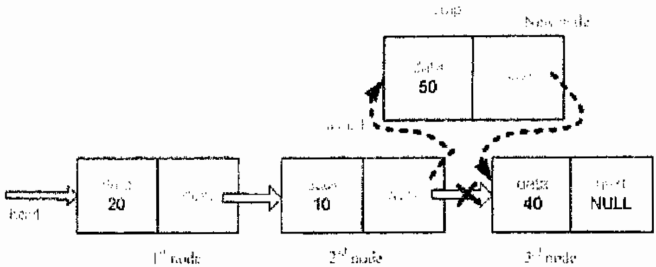
ولتفيذ عملية الربط بين العقدة الجديدة والقائمة المتصلة نفذ التعليمات التالية:

```
temp->next = temp1->next;

   //transfer the address of temp1->next to temp->next

temp1->next = temp;
```

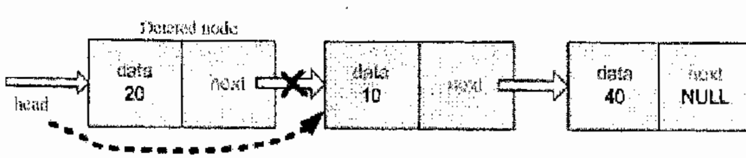
//transfer the address of temp to temp1->next



Linked list

للحذف من بداية القائمة نفذ التعليمات التالية:

```
node *temp;
                    // create a temporary node
temp =new node;
  // allocate space for node
temp = head;
             // transfer the address of 'head' to 'temp'
head = temp->next;
  // transfer the address of 'temp->next' to 'head'
delete(temp);
```



Linked list

اما لتفيذ عملية الحذف من نهاية القائمة فيمكن تنفيذ التعليمات التالية:

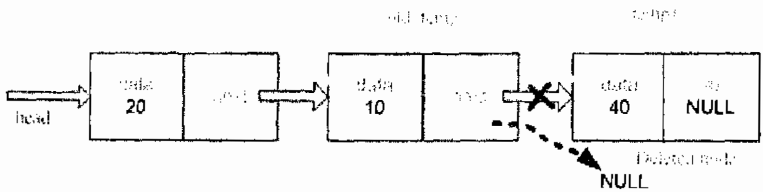
```
// crcate a temporary node
node *temp1;
temp1 =new node;
 // allocate space for node
temp1 = head;
                //transfer the address of head to temp1
node *old_temp;
              // create a temporary node
old_temp =new node;
   // allocate space for node


while(temp1->next!=NULL)          // go to the last node
{
    old_temp = temp1;
// transfer the address of 'temp1' to 'old_temp'
    temp1 = temp1->next;
   // transfer the address of 'temp1->next' to 'temp1'
}
```

والآن وبعدما اصبح المؤشر يشير الى العقدة المطلوبة نفذ ما يلي:

```
old_temp->next = NULL;
      // previous node of the last node is null
```

```
delete(temp1);
```



Linked list

لحذف عقدة محددة نفذ ما يلي:

```
node *temp1;

                // create a temporary node

temp1 =new node;

 // allocate space for node

temp1 = head;

          // transfer the address of 'head' to 'temp1'


node *old_temp;

  // create a temporary node

old_temp = new node;

   // allocate space for node

old_temp = temp1;

   // transfer the address of 'temp1' to 'old_temp'

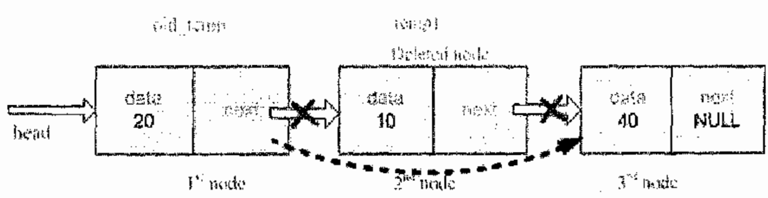
cout<<"ENTER THE NODE NUMBER:";

cin>>node_number;
```

```
            // take location
for( int i = 1 ; i < node_number ; i++ )
{
     old_temp = temp1;
              // store previous node
     temp1 = temp1->next;
          // store current node
}
```

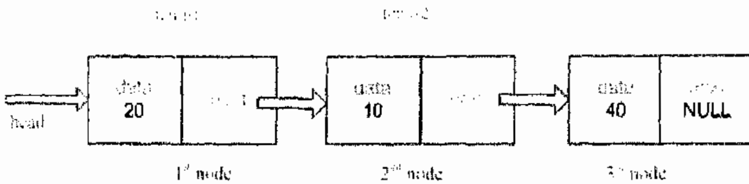
والان فان المؤشر node *temp1 سوف يشير الى العقدة المراد حذفها اما المؤشر old_temp* فانه سيكون مشيرا الى العقدة السابقة:

```
old_temp->next = temp1->next;
// transfer the address of 'temp1->next' to 'old_temp->next'
delete(temp1);
```



Linked list

لترتيب عناصر القائمة التالية ترتيبا تصاعديا:



Linked list

نفذ التالي:

```
node *temp1;
                // create a temporary node
temp1 = new node ;
 // allocate space for node


node *temp2;
         // create a temporary node
temp2 =new node ;
 // allocate space for node


int temp = 0;
                // store temporary data value


for( temp1 = head ; temp1!=NULL ; temp1 = temp1->next )
{
```

```
  for( temp2 = temp1->next ; temp2!=NULL ; temp2 = temp2->next )
    {
        if( temp1->data > temp2->data )
        {
            temp = temp1->data;
            temp1->data = temp2->data;
            temp2->data = temp;
        }
    }
}
```

وفيما يلي برنامج يستخدم تركيبة معرفة داخل الصنف للاعلان عن عنصر القائمة المتصلة حيث ينفذ هذا البرنامج كافة العمليات التي اشرنا اليها في هذه الوحدة:

```
#include <iostream>

using namespace std;

class linklist
{
    private:

```

```
        struct node
    {
        int data;
        node *link;
    }*p;

  public:

        linklist();
    void append( int num );
    void add_as_first( int num );
    void addafter( int c, int num );
    void del( int num );
    void display();
    int count();
    ~linklist();
};

linklist::linklist()
{
  p=NULL;
```

```
}

void linklist::append(int num)
{
   node *q,*t;

   if( p == NULL )
   {
      p = new node;
      p->data = num;
      p->link = NULL;
   }
   else
   {
      q = p;
      while( q->link != NULL )
         q = q->link;

      t = new node;
      t->data = num;
      t->link = NULL;
```

```
        q->link = t;
    }
}

void linklist::add_as_first(int num)
{
    node *q;

    q = new node;
    q->data = num;
    q->link = p;
    p = q;
}

void linklist::addafter( int c, int num)
{
    node *q,*t;
    int i;
    for(i=0,q=p;i<c;i++)
    {
        q = q->link;
```

```
    if( q == NULL )
    {
        cout<<"\nThere are less than "<<c<<" elements.";
       return;
    }
}

t = new node;
t->data = num;
t->link = q->link;
q->link = t;
}

void linklist::del( int num )
{
   node *q,*r;
   q = p;
   if( q->data == num )
   {
       p = q->link;
      delete q;
```

```
    return;
}

r = q;
while( q!=NULL )
{
    if( q->data == num )
    {
        r->link = q->link;
        delete q;
        return;
    }

    r = q;
    q = q->link;
  }
  cout<<"\nElement "<<num<<" not Found.";
}

void linklist::display()
{
```

```
      node *q;
    cout<<endl;

    for( q = p ; q != NULL ; q = q->link )
        cout<<endl<<q->data;

}

int linklist::count()
{
      node *q;
    int c=0;
    for( q=p ; q != NULL ; q = q->link )
        c++;

    return c;
}

linklist::~linklist()
{
      node *q;
```

```
    if( p == NULL )
        return;

    while( p != NULL )
    {
        q = p->link;
      delete p;
      p = q;
    }
}

int main()
{
    linklist ll;
   cout<<"No. of elements = "<<ll.count();
   ll.append(12);
   ll.append(13);
   ll.append(23);
   ll.append(43);
   ll.append(44);
   ll.append(50);
```

```
ll.add_as_first(2);
ll.add_as_first(1);

ll.addafter(3,333);
ll.addafter(6,666);

ll.display();
cout<<"\nNo. of elements = "<<ll.count();

ll.del(333);
ll.del(12);
ll.del(98);
cout<<"\nNo. of elements = "<<ll.count();
return 0;
}
```

والبرنامج التالي يستخدم قائمة متصلة يتكون كل عنصر فيها من متغير رمزي ومتغير صحيح ومتغير كسري بالإضافة الى المؤشر:

```
#include <iostream.h>

struct node
 { char name[20];   // Name of up to 20 letters
   int age;         // D.O.B. would be better
```

```
    float height;     // In metres
    node *nxt;// Pointer to next node
  };

node *start_ptr = NULL;
node *current;               // Used to move along the list
int option = 0;

void add_node_at_end()
  { node *temp, *temp2;  // Temporary pointers

    // Reserve space for new node and fill it with data
    temp = new node;
    cout << "Please enter the name of the person: ";
    cin >> temp->name;
    cout << "Please enter the age of the person: ";
    cin >> temp->age;
    cout << "Please enter the height of the person: ";
    cin >> temp->height;
    temp->nxt = NULL;

    // Set up link to this node
    if (start_ptr == NULL)
      { start_ptr = temp;
            current = start_ptr;
      }
    else
      { temp2 = start_ptr;
       // We know this is not NULL - list not empty!
       while (temp2->nxt != NULL)
         {  temp2 = temp2->nxt;
           // Move to next link in chain
         }
       temp2->nxt = temp;
      }
  }

void display_list()
```

```
{ node *temp;
  temp = start_ptr;
  cout << endl;
  if (temp == NULL)
   cout << "The list is empty!" << endl;
  else
   { while (temp != NULL)
        { // Display details for what temp points to
       cout << "Name: " << temp->name << " ";
       cout << "Age: " << temp->age << " ";
          cout << "Height: " << temp->height;
          if (temp == current)
              cout << " <-- Current node";
       cout << endl;
          temp = temp->nxt;

        }
       cout << "End of list!" << endl;
   }
}

void delete_start_node()
 { node *temp;
  temp = start_ptr;
  start_ptr = start_ptr->nxt;
  delete temp;
 }

void delete_end_node()
 { node *temp1, *temp2;
  if (start_ptr == NULL)
     cout << "The list is empty!" << endl;
  else
    { temp1 = start_ptr;
     if (temp1->nxt == NULL)
       { delete temp1;
        start_ptr = NULL;
       }
```

```
        else
          { while (temp1->nxt != NULL)
            { temp2 = temp1;
              temp1 = temp1->nxt;
            }
           delete temp1;
           temp2->nxt = NULL;
          }
      }
  }

void move_current_on ()
  { if (current->nxt == NULL)
      cout << "You are at the end of the list." << endl;
     else
      current = current->nxt;
  }

void move_current_back ()
  { if (current == start_ptr)
      cout << "You are at the start of the list" << endl;
     else
      { node *previous;     // Declare the pointer
        previous = start_ptr;

        while (previous->nxt != current)
          { previous = previous->nxt;
          }
        current = previous;
      }
  }

void main()
 {  start_ptr = NULL;
    do
         {
          display_list();
          cout << endl;
```

```
        cout << "Please select an option: " << endl;
        cout << "0. Exit the program." << endl;
        cout << "1. Add a node to the end of the list." << endl;
        cout << "2. Delete the start node from the list." << endl;
        cout << "3. Delete the end node from the list." << endl;
        cout << "4. Move the current pointer on one node." <<
endl;
     cout << "5. Move the current pointer back one node." <<
endl;
     cout << endl << " >> ";
        cin >> option;

        switch (option)
          {
           case 1: add_node_at_end(); break;
           case 2: delete_start_node(); break;
           case 3: delete_end_node(); break;
           case 4: move_current_on(); break;
       case 5: move_current_back();
          }
       }
  while (option != 0);
 }
```

يمكن استخدام المؤشر للاشارة الى الصنف ولتوضيح هذا لناخذ البرنامج التالي:

```
// pointer to classes example
#include <iostream>
using namespace std;

class CRectangle {
  int width, height;
 public:
  void set_values (int, int);
  int area (void) {return (width * height);}
};

void Crectangle::set_values (int a, int b) {
```

```
  width = a;
  height = b;
}

int main () {
  Crectangle a, *b, *c;
  Crectangle * d = new Crectangle[2];
  b= new Crectangle;
  c= &a;
  a.set_values (1,2);
  b->set_values (3,4);
  d->set_values (5,6);
  d[1].set_values (7,8);
  cout << "a area: " << a.area() << endl;
  cout << "*b area: " << b->area() << endl;
  cout << "*c area: " << c->area() << endl;
  cout << "d[0] area: " << d[0].area() << endl;
  cout << "d[1] area: " << d[1].area() << endl;
  delete[] d;
  delete b;
  return 0;
}
```

تم في هذا البرنامج تعريف 4 اهداف باستخدام الصنف المعلن عنه الأول تم تعريفه بالاسم والثلاثة الاخرى باستخدام المؤشرات والتي يشير كل منها الى هدف من نوع الصنف المعلن عنه في البرنامج ولو نفذنا هذا البرنامج فإننا سنحصل على النتيجة التالية:

```
a area: 2
*b area: 12
*c area: 2
d[0] area: 30
d[1] area: 56
```

معاملات الافراط في التحميل:

Overloading operators

لناخذ التعبير الحسابي التالي:

```
int a, b, c;
a = b + c;
```

واضح ان هذا التعبير صحيح في لغة سي بلس بلس كون كافة المتغيرات الداخلة في التعبير متشابهه وعند ترجمته من قبل المترجم فانه لن يعطي اي خطا.

لناخذ الان التركيبة التالية:

```
struct {
  string product;
  float price;
} a, b, c;
a = b + c;
```

لو ادخلنا هذه التركيبة في برنامج سي بلس بلس بنفس الصورة المبينة اعلاه فان المترجم سيعطينا خطا وذلك لاننا لم نحدد خصائص عملية الجمع في مجموعة التعليمات اعلاه.

تمتلك لغة سي بلس بلس امكانية التعامل مع معامل الجمع وغيره من معاملات باستخدام مفهوم معاملات كثرة التحميل والتي تاخذ الصورة التالية عند تعريفها:

```
type operator sign (parameters) { /*...*/ }
```

ويبين الجدول التالي هذه المعاملات:

Overloadable operators														
+	-	*	/	=	<	>	+=	-=	*=	/=	<<	>>		
<<=	>>=	==	!=	<=	>=	++	--	%	&	^	!	|		
~	&=	^=	|=	&&	||	%=	[]	()	,	->*	->	new		
delete	new[]	delete[]												

والبرنامج التالي يبين كيفية استخدام معامل الجمع:

```
// vectors: overloading operators example
#include <iostream>
using namespace std;

class CVector {
  public:
    int x,y;
    CVector () {};
    CVector (int,int);
    CVector operator + (CVector);
};

CVector::CVector (int a, int b) {
  x = a;
  y = b;
}

CVector CVector::operator+ (CVector param) {
  CVector temp;
  temp.x = x + param.x;
  temp.y = y + param.y;
  return (temp);
}

int main () {
  CVector a (3,1);
  CVector b (1,2);
```

```
  CVector c;
  c = a + b;
  cout << c.x << "," << c.y;
  return 0;
}
```

4,3

والجدول التالي يبين كيفية استخدام هذه المعاملات داخل الصنف وكاقترانات اعضاء:

Expression	Operator	Member function	Global function
@a	+ - * & ! ~ ++ --	A::operator@()	operator@(A)
a@	++ --	A::operator@(int)	operator@(A,int)
a@b	+ - * / % ^ & | < > == != <= >= << >> && || ,	A::operator@ (B)	operator@(A,B)
a@b	= += -= *= /= %= ^= &= |= <<= >>= []	A::operator@ (B)	-
a(b, c...)	()	A::operator() (B, C...)	-
a->x	->	A::operator->()	-

يتعامل الصنف مع ما يسمى باعضاء البيانات الاستاتيكية والتي يطلق عليها ايضا متغيرات الصنف وكمثال على ذلك ندرج البرنامج التالي والذي يستخدم العضو الاستاتيكي لتعداد عدد الاهداف المعرفة من قبل الصنف:

```
// static members in classes
#include <iostream>
using namespace std;

class CDummy {
  public:
    static int n;
    CDummy () { n++; };
    ~CDummy () { n--; };
```

```
};

int CDummy::n=0;

int main () {
  CDummy a;
  CDummy b[5];
  CDummy * c = new CDummy;
  cout << a.n << endl;
  delete c;
  cout << CDummy::n << endl;
  return 0;
}
```

```
7
6
```

الوحدة الرابعة

4

الأصناف المشتقة

الاصناف المشتقة

كما اشرنا سابقا فان الصنف قد يتضمن انواعاً متعددة من البيانات وقد تشتمل هذه الاعضاء ايضا على صنف وفي هذه الحالة يعرف الصنف المضمن في صنف اخر بالصنف المشتق وياخذ الصنف المشتق الشكل التالي:

```
class derived-class:access-specifier base-class
 {
  ...
  ...
  ...
 };
```

للاصناف المشتقة اهمية كبيرة حيث تستخدم لاختصار البرنامج وتسهيل عملية التعامل مع البيانات والاقترانات المختلفة وعلى سبيل المثال لناخذ الصنفين التاليين:

```
class computer
 {
  int speed;
  int main_memory;
  int harddisk_memory;

 public:
  void set_speed(int);
  void set_mmemory(int);
  void set_hmemory(int);
  int get_speed();
  int get_mmemory();
  int get_hmemory();
 };
```

```
class laptop
 {
  int speed;
  int main_memory;
  int harddisk_memory;

  int battery_time;
  float weight;

 public:
  void set_speed(int);
  void set_mmemory(int);
  void set_hmemory(int);
  int get_speed();
  int get_mmemory();
  int get_hmemory();

  void set_battime(int);
  void set_weight(float);
  int get_battime();
  float get_weight();
 };
```

لاحظ ان الصنفين يشتركان في مجموعة من البيانات ومجموعة من الاقترانات وعليه يمكن اخذ البيانات المشتركة واتباعها للصنف الاصيل حيث يتم توريثها الى الصنف المشتق والذي يعرف كعضو من اعضاء الصنف الاصيل ولاعادة كتابة الاصناف السابقة باستخدام مفهوم الصنف الاصيل والمشتق فاننا نحصل على ما يلي:

```
class computer// base class
 {
```

```
    int speed;
    int main_memory;
    int harddisk_memory;

  public:
    void set_speed(int);
    void set_mmemory(int);
    void set_hmemory(int);
    int get_speed();
    int get_mmemory();
    int get_hmemory();
  };
class laptop:public computer//derived class
  {
    int battery_time;
    float weight;

  public:
    void set_battime(int);
    void set_weight(float);
    int get_battime();
    float get_weight();
  };
```

والان يمكننا التعامل مع هذين الصنفين الصنف الاساسي والصنف المشتق تماما كما تعملنا مع الصنف الاساسي والبرنامج التالي يبين كيفية التعامل مع الصنف المشتق والاساسي:

```
// Introduction to Inheritance in C++
 // ----------------------
 // An example program to
```

```
// demonstrate inheritance in C++
#include<iostream.h>

// base class for inheritance
class computer
{
 float speed;
 int main_memory;
 int harddisk_memory;

public:
 void set_speed(float);
 void set_mmemory(int);
 void set_hmemory(int);
 float get_speed();
 int get_mmemory();
 int get_hmemory();
};

// -- MEMBER FUNCTIONS --
void computer::set_speed(float sp)
{
 speed=sp;
}

void computer::set_mmemory(int m)
{
 main_memory=m;
}

void computer::set_hmemory(int h)
{
```

```
  harddisk_memory=h;
 }

 int computer::get_hmemory()
 {
  return harddisk_memory;
 }

 int computer::get_mmemory()
 {
  return main_memory;
 }

 float computer::get_speed()
 {
  return speed;
 }
// --   ENDS   --

// inherited class
class laptop:public computer
{
 int battery_time;
 float weight;

public:
 void set_battime(int);
 void set_weight(float);
 int get_battime();
 float get_weight();
};
```

```
// -- MEMBER FUNCTIONS --
void laptop::set_battime(int b)
{
 battery_time=b;
}

void laptop::set_weight(float w)
{
 weight=w;
}

int laptop::get_battime()
{
 return battery_time;
}

float laptop::get_weight()
{
 return weight;
}
// --   ENDS   --

void main(void)
{
 computer c;
 laptop l;

 c.set_mmemory(512);
 c.set_hmemory(1024);
 c.set_speed(3.60);
```

```
// set common features
l.set_mmemory(256);
l.set_hmemory(512);
l.set_speed(1.8);

// set specific features
l.set_battime(7);
l.set_weight(2.6);

// show details of base class object
cout<<"Info. of computer class\n\n";
cout<<"Speed:"<<c.get_speed()<<"\n";
cout<<"Main Memory:"<<c.get_mmemory()<<"\n";
cout<<"Hard Disk Memory:"<<c.get_hmemory()<<"\n";

//show details of derived class object
cout<<"\n\nInfo. of laptop class\n\n";
cout<<"Speed:"<<l.get_speed()<<"\n";
cout<<"Main Memory:"<<l.get_mmemory()<<"\n";
cout<<"Hard Disk Memory:"<<l.get_hmemory()<<"\n";
cout<<"Weight:"<<l.get_weight()<<"\n";
cout<<"Battery Time:"<<l.get_battime()<<"\n";
}
```

الاقتران الصديق:

كما اشرنا سابقا فان الاعضاء الخاصة والاعضاء المحمية لا يمكن الوصول اليها من خارج الصنف المعرفة فيه لكن هذه القاعدة لا تنطبق على الاقتران الصديق او الصنف الصديق. فالاقتران الصديق ما هو الا اقتران خارجي يتم الاعلان عنه باستخدام الكلمة المجوزة "صديق" بحيث يستطيع هذا الاقتران الوصول الى الاعضاء الخاصة والمحمية في الصنف والمثال التالي يبين كيفية الاعلان عن الاقتران الصديق واستخدامه:

```
// friend functions
#include <iostream>
using namespace std;

class CRectangle {
   int width, height;
  public:
   void set_values (int, int);
   int area () {return (width * height);}
   friend CRectangle duplicate (CRectangle);
};

void CRectangle::set_values (int a, int b) {
  width = a;
  height = b;
}

CRectangle duplicate (CRectangle rectparam)
{
  CRectangle rectres;
  rectres.width = rectparam.width*2;
  rectres.height = rectparam.height*2;
  return (rectres);
}

int main () {
  CRectangle rect, rectb;
  rect.set_values (2,3);
  rectb = duplicate (rect);
  cout << rectb.area();
  return 0;
}
```

24

اضافة لذلك بإمكاننا ايضا تعريف الصنف الصديق والذي يستطيع الوصول الى الاعضاء الخاصة والمحمية في الصنف الاساسي والمثال التالي يبين

كيفية الاعلان عن الصنف الصديق وكيفيية تمكينه من استخدام اعضاء الصنف الاساسي:

```
// friend class
#include <iostream>
using namespace std;

class CSquare;

class CRectangle {
   int width, height;
  public:
   int area ()
     {return (width * height);}
   void convert (CSquare a);
};

class CSquare {
  private:
   int side;
  public:
   void set_side (int a)
     {side=a;}
   friend class CRectangle;
};

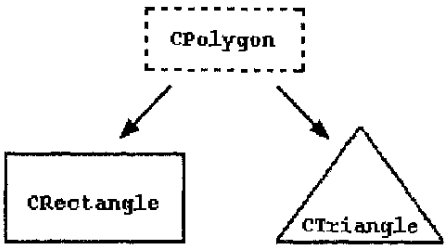
void CRectangle::convert (CSquare a) {
  width = a.side;
  height = a.side;
}

int main () {
  CSquare sqr;
  CRectangle rect;
  sqr.set_side(4);
  rect.convert(sqr);
  cout << rect.area();
```

```
 return 0;
}
```

16

لناخذ الاصناف التالية والمبينة في الشكل التالي:



من خلال الشكل يتبين لنا ان الصنف الاساسي مرتبط مع الاصناف الاخرى بعلاقة توريث ولناخذ هذه الاصناف كصنف اساسي واصناف مشتقة وكما هو مبين ادناه:

```
// derived classes
#include <iostream>
using namespace std;

class CPolygon {
 protected:
  int width, height;
 public:
  void set_values (int a, int b)
    { width=a; height=b;}
 };

class CRectangle: public CPolygon {
 public:
  int area ()
    { return (width * height); }
```

```
};

class CTriangle: public CPolygon {
  public:
    int area ()
      { return (width * height / 2); }
  };


int main () {
  CRectangle rect;
  CTriangle trgl;
  rect.set_values (4,5);
  trgl.set_values (4,5);
  cout << rect.area() << endl;
  cout << trgl.area() << endl;
  return 0;
}
```

```
20
10
```

لاحظ ان عملية توريث الاعضاء من الصنف الاساسي الى الاصناف المشتقة بناء على المعلومات المبينة في الجدول ادناه:

Access	public	protected	private
members of the same class	yes	yes	yes
members of derived classes	yes	yes	no
not members	yes	no	no

هذا ويمكن للصنف الاساسي توريث عناصر التهيئة او البناء وعناصر الهدم الى الاصناف المشتقة منه ولبيان ذلك لنأخذ البرنامج التالي:

```
// constructors and derived classes
#include <iostream>
using namespace std;
```

```
class mother {
  public:
    mother ()
      { cout << "mother: no parameters\n"; }
    mother (int a)
      { cout << "mother: int parameter\n"; }
};

class daughter: public mother {
  public:
    daughter (int a)
      { cout << "daughter: int parameter\n\n"; }
};

class son: public mother {
  public:
    son (int a): mother (a)
      { cout << "son: int parameter\n\n"; }
};

int main () {
  daughter cynthia (0);
  son daniel(0);

  return 0;
}
```

لاحظ ان الصنف المشتق الاول نفذ عنصر التهيئة المرجعي من الصنف الاساسي وعنصر التهيئة الخاص به اما الصنف المشتق الثاني فقد نفذ عنصر التهيئة على شكل الاقتران من الصنف الاساسي وعنصر التهيئة الخاص به وذلك لان عنصر التهيئة المرجعي ينفذ مرة واحدة وعليه تكون نتيجة التنفيذ كما يلي:

```
mother: no parameters
daughter: int parameter

mother: int parameter
son: int parameter
```

والامثلة التالية تبين كيفية التعامل مع عناصر البناء والهدم في الاصناف:

```
#include <iostream>
using namespace std;

class BaseClass1 {
public:
 BaseClass1() { cout << "Constructing BaseClass1\n"; }
 ~BaseClass1() { cout << "Destructing BaseClass1\n"; }
};

class BaseClass2 {
public:
 BaseClass2() { cout << "Constructing BaseClass2\n"; }
 ~BaseClass2() { cout << "Destructing BaseClass2\n"; }
};

class DerivedClass: public BaseClass1, public BaseClass2 {
public:
 DerivedClass() { cout << "Constructing DerivedClass\n"; }
 ~DerivedClass() { cout << "Destructing DerivedClass\n"; }
};

int main()
{
 DerivedClass ob;

 return 0;
}
```

```
Constructing BaseClass1
Constructing BaseClass2
Constructing DerivedClass
Destructing DerivedClass
Destructing BaseClass2
Destructing BaseClass1
```

```
#include <iostream>
using namespace std;

class BaseClass1 {
protected:
 int i;
public:
 BaseClass1(int x) {
   i =x;
   cout << "Constructing BaseClass1\n";
 }
 ~BaseClass1() {
   cout << "Destructing BaseClass1\n";
 }
};

class BaseClass2 {
protected:
 int k;
public:
 BaseClass2(int x) {
   k = x;
   cout << "Constructing base2\n";
 }
 ~BaseClass2() {
   cout << "Destructing base2\n";
 }
};

class DerivedClass: public BaseClass1, public BaseClass2 {
```

```
public:
  DerivedClass(int x, int y): BaseClass1(x), BaseClass2(y) {
    cout << "Constructing DerivedClass\n";
  }

  ~DerivedClass() {
    cout << "Destructing DerivedClass\n";
  }
  void show() {
    cout << i << " " << k << endl;
  }
};

int main()
{
  DerivedClass ob(3, 4);

  ob.show();

  return 0;
}
```

```
Constructing BaseClass1
Constructing base2
Constructing DerivedClass
3 4
Destructing DerivedClass
Destructing base2
Destructing BaseClass1
```

قد يرث الصنف اعضاء من اكثر من صنف اساسي وفي هذه الحالة يتم التعامل مع عملية التوريث وكما اشرنا اليها سابقا والمثال التالي يوضح آلية تنفيذ عملية التوريث المتعددة:

```
// multiple inheritance
#include <iostream>
```

```
using namespace std;

class CPolygon {
 protected:
  int width, height;
 public:
  void set_values (int a, int b)
   { width=a; height=b;}
 };

class COutput {
 public:
  void output (int i);
 };

void COutput::output (int i) {
 cout << i << endl;
 }

class CRectangle: public CPolygon, public COutput {
 public:
  int area ()
   { return (width * height); }
 };

class CTriangle: public CPolygon, public COutput {
 public:
  int area ()
   { return (width * height / 2); }
 };

int main () {
 CRectangle rect;
 CTriangle trgl;
 rect.set_values (4,5);
 trgl.set_values (4,5);
 rect.output (rect.area());
 trgl.output (trgl.area());
```

```
 return 0;
}
```

```
20
10
```

الاعضاء الافتراضية:

الاعضـاء الافتراضـية في الصـنف الاساسـي هـي اعضاء يعـاد تعريفهـا في الصـنف المشـتق بطريقـة جديـدة ويـتم الاعـلان عنهـا باسـتخدام الكلمـة المحجـوزة "افتراضي".

والبرنـامج التـالي يـبين كيفيـة الاعـلان عـن اقـتران افتراضـي وكيفيـة استخدامه:

```
class Window // Base class for C++ virtual function example
   {
    public:
      virtual void Create()

// virtual function for C++ virtual function example
      {
          cout <<"Base class Window";
      }
   };

   class CommandButton: public Window
   {
    public:
      void Create()
      {
 cout<<"Derived class Command Button “;
      }
   };

   void main()
   {
```

```
      Window  *x, *y;

      x = new Window();
      x->Create();

      y = new CommandButton();
      y->Create();
   }
```

The output of the above program will be,

Base class Window

Derived class Command Button

مثال اخر:

```
//Virtual function for two derived classes

#include <iostream>

using namespace std;



class figure {

protected:

 double x, y;

public:

 void set_dim(double i, double j) {

  x = i;

  y = j;

 }

 virtual void show_area() {
```

```
    cout << "No area computation defined ";
    cout << "for this class.\n";
  }
};

class triangle: public figure {
  public:
    void show_area() {
      cout << "Triangle with height ";
      cout << x << " and base " << y;
      cout << " has an area of ";
      cout << x * 0.5 * y << ".\n";
    }
};

class rectangle: public figure {
  public:
    void show_area() {
      cout << "Rectangle with dimensions ";
      cout << x << "x" << y;
      cout << " has an area of ";
```

```
    cout << x *  y << ".\n";
  }
};

int main()
{
 figure *p; // create a pointer to base type

 triangle t; // create objects of derived types
 rectangle s;

 p = &t;
 p->set_dim(10.0, 5.0);
 p->show_area();

 p = &s;
 p->set_dim(10.0, 5.0);
 p->show_area();
  return 0;
}
```

مثال:

```
#include <iostream>
using namespace std;
class Animal
{
        public:
        virtual void eat()
        {
                cout<<"I'm an animal"<<endl;
        }
};
class Dog: public Animal
{
        public:
        void eat()
        {
                cout<<"I eat like a dog"<<endl;
        }
};
class Cat: public Animal
{
```

```
        public:
        void eat()
        {
                cout<<"I eat like a cat"<<endl;
        }
};
void test ( Animal & a);
int main()
{
        Animal a;
        Dog b;
        Cat c;
        test ( a );
        test ( b );
        test ( c );
        return 0;
}
void test ( Animal & a)
{
        a.eat();
}
```

./Temp

I'm an animal

I eat like a dog

I eat like a cat

مثال:

```
#include <iostream>
using namespace std;

class CPolygon {
 protected:
  int width, height;
 public:
  void set_values (int a, int b)
    { width=a; height=b; }
  virtual int area (void) =0;
 };

class CRectangle: public CPolygon {
 public:
  int area (void)
    { return (width * height); }
 };

class CTriangle: public CPolygon {
 public:
  int area (void)
    { return (width * height / 2); }
 };

int main () {
 CRectangle rect;
 CTriangle trgl;
 CPolygon * ppoly1 = &rect;
```

```
  CPolygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  cout << ppoly1->area() << endl;
  cout << ppoly2->area() << endl;
  return 0;
}
```

```
20
10
```

مثال:

```
// pure virtual members can be called
// from the abstract base class
#include <iostream>
using namespace std;

class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    virtual int area (void) =0;
    void printarea (void)
      { cout << this->area() << endl; }
  };

class CRectangle: public CPolygon {
  public:
    int area (void)
      { return (width * height); }
  };

class CTriangle: public CPolygon {
  public:
```

```
  int area (void)
    { return (width * height / 2); }
 };


int main () {
 CRectangle rect;
 CTriangle trgl;
 CPolygon * ppoly1 = &rect;
 CPolygon * ppoly2 = &trgl;
 ppoly1->set_values (4,5);
 ppoly2->set_values (4,5);
 ppoly1->printarea();
 ppoly2->printarea();
 return 0;
}
20
10
```

مثال:

```
#include <iostream>
using namespace std;

class CPolygon {
 protected:
  int width, height;
 public:
  void set_values (int a, int b)
    { width=a; height=b; }
  virtual int area (void) =0;
  void printarea (void)
    { cout << this->area() << endl; }
 };

class CRectangle: public CPolygon {
 public:
  int area (void)
```

```
    { return (width * height); }
  };

class CTriangle: public CPolygon {
  public:
    int area (void)
      { return (width * height / 2); }
  };

int main () {
  CPolygon * ppoly1 = new CRectangle;
  CPolygon * ppoly2 = new CTriangle;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  ppoly1->printarea();
  ppoly2->printarea();
  delete ppoly1;
  delete ppoly2;
  return 0;
}
```

```
20
10
```

مثال:

```
#include <string.h>
    #include <assert.h>
    #include <iostream.h>

    typedef double Coord;

    /*
    The type of X/Y points on the screen.
    */

    enum Color {Co_red, Co_green, Co_blue};
```

```
/*
Colors.
*/

// abstract base class for all shape types
class Shape {
protected:
      Coord xorig; // X origin
      Coord yorig; // Y origin
      Color co; // color

/*
These are protected so that they can be accessed
by derived classes.  Private wouldn't allow this.

These data members are common to all shape types.
*/

public:
      Shape(Coord x, Coord y, Color c):
         xorig(x), yorig(y), co(c) {} // constructor

/*
Constructor to initialize data members common to
all shape types.
*/

      virtual ~Shape() {} // virtual destructor

/*
Destructor for Shape.  It's a virtual function.
Destructors in derived classes are virtual also
because this one is declared so.
*/

      virtual void draw() = 0; // pure virtual draw() function

/*
```

```
Similarly for the draw() function.  It's a pure virtual and
is not called directly.
*/

};

// line with X,Y destination
class Line: public Shape {

/*
Line is derived from Shape, and picks up its
data members.
*/

      Coord xdest; // X destination
      Coord ydest; // Y destination

/*
Additional data members needed only for Lines.
*/

public:
      Line(Coord x, Coord y, Color c, Coord xd, Coord yd):
         xdest(xd), ydest(yd),
         Shape(x, y, c) {} // constructor with base initialization

/*
Construct a Line, calling the Shape constructor as well
to initialize data members of the base class.
*/

      ~Line() {cout << "~Line\n";} // virtual destructor

/*
Destructor.
*/

      void draw() // virtual draw function
```

```
    {
        cout << "Line" << "(";
        cout << xorig << ", " << yorig << ", " << int(co);
        cout << ", " << xdest << ", " << ydest;
        cout << ")\n";
    }

/*
Draw a line.
*/

};

// circle with radius
class Circle: public Shape {
    Coord rad; // radius of circle

/*
Radius of circle.
*/

public:
    Circle(Coord x, Coord y, Color c, Coord r): rad(r),
      Shape(x, y, c) {} // constructor with base initialization

    ~Circle() {cout << "~Circle\n";} // virtual destructor
    void draw() // virtual draw function
    {
        cout << "Circle" << "(";
        cout << xorig << ", " << yorig << ", " << int(co);
        cout << ", " << rad;
        cout << ")\n";
    }
};

// text with characters given
class Text: public Shape {
    char* str; // copy of string
```

```
public:
    Text(Coord x, Coord y, Color c, const char* s):
      Shape(x, y, c) // constructor with base initialization
    {
        str = new char[strlen(s) + 1];
        assert(str);
        strcpy(str, s);

/*
Copy out text string. Note that this would be done differently
if we were taking advantage of some newer C++ features like
exceptions and strings.
*/

    }
    ~Text() {delete [] str; cout << "~Text\n";} // virtual dtor

/*
Destructor; delete text string.
*/

    void draw() // virtual draw function
    {
        cout << "Text" << "(";
        cout << xorig << ", " << yorig << ", " << int(co);
        cout << ", " << str;
        cout << ")\n";
    }
};

int main()
{
    const int N = 5;
    int i;
    Shape* sptrs[N];

/*
Pointer to vector of Shape* pointers. Pointers to classes
```

```
derived from Shape can be assigned to Shape* pointers.
*/

      // initialize set of Shape object pointers

      sptrs[0] = new Line(0.1, 0.1, Co_blue, 0.4, 0.5);
      sptrs[1] = new Line(0.3, 0.2, Co_red, 0.9, 0.75);
      sptrs[2] = new Circle(0.5, 0.5, Co_green, 0.3);
      sptrs[3] = new Text(0.7, 0.4, Co_blue, "Howdy!");
      sptrs[4] = new Circle(0.3, 0.3, Co_red, 0.1);

/*
Create some shape objects.
*/

      // draw set of shape objects

      for (i = 0; i < N; i++)
            sptrs[i]->draw();

/*
Draw them using virtual functions to pick up the
right draw() function based on the actual object
type being pointed at.
*/

      // cleanup

      for (i = 0; i < N; i++)
            delete sptrs[i];

/*
Clean up the objects using virtual destructors.
*/

      return 0;
}
```

When we run this program, the output is:

```
Line(0.1, 0.1, 2, 0.4, 0.5)
Line(0.3, 0.2, 0, 0.9, 0.75)
Circle(0.5, 0.5, 1, 0.3)
Text(0.7, 0.4, 2, Howdy!)
Circle(0.3, 0.3, 0, 0.1)
~Line
~Line
~Circle
~Text
~Circle
```

الوحدة الخامسة

5

القوالب

Templates

القوالب
Templates

تعتبر اقترانات القوالب من الاقترانات الخاصة والتي يمكن ان تتعامل مع أنواع مختلفة من البيانات والذي يمكننا من استحداث اقتران يمكن في المستقبل ملائمته مع اي نوع من انواع البيانات او الاصناف بدون الحاجة الى تغيير محتوى الاقتران من تعليمات وفي لغة سي بلس بلس يمكن تحقيق هذا باستخدام اقترانات القوالب.

ان معلم القالب هو معلم خاص يمكن استخدامه لتمرير نوع البيانات تماما كما يتم تمرير قيمة المعلم والصيغة العامة للاعلان عن اقتران القوالب تاخذ الشكل التالي:

```
template <class identifier> function_declaration;
template <typename identifier> function_declaration;
```

والفرق الوحيد بين الصيغتين هو استخدام اما الصنف او نوع البيانات.

فمثلا لاستحداث اقتران قالب لارجاع القيمة الكبرى من بين قيم هدفين يمكن تنفيذ ما يلي:

```
template <class myType>
myType GetMax (myType a, myType b) {
 return (a>b?a:b);
}
```

استحدثنا هنا اقتران باستخدام نوع البيانات كقالب ولاستخدام هذا الاقتران بالصيغة التالية:

```
function_name <type> (parameters);
```

ولاستدعاء هذا الاقتران يمكن تنفيذ ما يلي:

```
int x,y;

GetMax <int> (x,y);
```

والبرنامج التالي يبين كيفية استخدام هذا الاقتران:

```
/ function template
#include <iostream>
using namespace std;

template <class T>
T GetMax (T a, T b) {
  T result;
  result = (a>b)? a: b;
  return (result);
}

int main () {
  int i=5, j=6, k;
  long l=10, m=5, n;
  k=GetMax<int>(i,j);
  n=GetMax<long>(l,m);
  cout << k << endl;
  cout << n << endl;
  return 0;
}
6
10
/ function template II
#include <iostream>
using namespace std;

template <class T>
T GetMax (T a, T b) {
  return (a>b?a:b);
```

```
}

int main () {
  int i=5, j=6, k;
  long l=10, m=5, n;
  k=GetMax(i,j);
  n=GetMax(l,m);
  cout << k << endl;
  cout << n << endl;
  return 0;
}
```

```
6
10
```

بإمكاننا ايضا استخدام الصنف كقالب وكما هو مبين في المثال التالي:

```
template <class T>
class mypair {
    T values [2];
  public:
    mypair (T first, T second)
    {
      values[0]=first; values[1]=second;
    }
};
```

```
// class templates
#include <iostream>
using namespace std;

template <class T>
class mypair {
    T a, b;
  public:
    mypair (T first, T second)
```

```
    {a=first; b=second;}
   T getmax ();
};

template <class T>
T mypair<T>::getmax ()
{
  T retval;
  retval = a>b? a: b;
  return retval;
}

int main () {
  mypair <int> myobject (100, 75);
  cout << myobject.getmax();
  return 0;
}
100
```

اذا اردنا تعريف عدة طرق لتنفيذ التعليمات فعلينا عند الاعلان تحديد تخصص القالب وكما هو مبين في البرنامج التالي:

```
// template specialization
#include <iostream>
using namespace std;

// class template:
template <class T>
class mycontainer {
    T element;
  public:
    mycontainer (T arg) {element=arg;}
    T increase () {return ++element;}
};

// class template specialization:
template <>
```

```
class mycontainer <char> {
   char element;
 public:
   mycontainer (char arg) {element=arg;}
   char uppercase ()
   {
     if ((element>='a')&&(element<='z'))
     element+='A'-'a';
     return element;
   }
};

int main () {
  mycontainer<int> myint (7);
  mycontainer<char> mychar ('j');
  cout << myint.increase() << endl;
  cout << mychar.uppercase() << endl;
  return 0;
}
8
J
```

والمثال التالي يبين كيفية استخدام القالب لمعالجة متجهات مختلفة في النوع:

```
// temlate to process difernt array

// Using template functions.
#include <iostream>
using std::cout;
using std::endl;

// function template printArray definition
template< typename T>
void printArray( const T *array, int count )
 {
    for ( int i = 0; i < count; i++ )
       cout << array[ i ] << " ";

```

```
   cout << endl;
} // end function template printArray

int main()
{
   const int ACOUNT = 5; // size of array a
   const int BCOUNT = 7; // size of array b
   const int CCOUNT = 6; // size of array c

   int a[ ACOUNT ] = { 1, 2, 3, 4, 5 };
   double b[ BCOUNT ] = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
   char c[ CCOUNT ] = "HELLO"; // 6th position for null

   cout << "Array a contains:" << endl;

   // call integer function-template specialization
   printArray( a, ACOUNT );

   cout << "Array b contains:" << endl;

   // call double function-template specialization
   printArray( b, BCOUNT );

   cout << "Array c contains:" << endl;

   // call character function-template specialization
printArray( c, CCOUNT );
   return 0;
   } // end main
```

```
Array a contains:
1 2 3 4 5
Array b contains:
1.1 2.2 3.3 4.4 5.5 6.6 7.7
Array c contains:
H E L L O
```

مثال:

```
// sequence template
#include <iostream>
using namespace std;

template <class T, int N>
class mysequence {
   T memblock [N];
  public:
   void setmember (int x, T value);
   T getmember (int x);
};

template <class T, int N>
void mysequence<T,N>::setmember (int x, T value) {
 memblock[x]=value;
}

template <class T, int N>
T mysequence<T,N>::getmember (int x) {
 return memblock[x];
}

int main () {
 mysequence <int,5> myints;
 mysequence <double,5> myfloats;
 myints.setmember (0,100);
 myfloats.setmember (3,3.1416);
 cout << myints.getmember(0) << '\n';
 cout << myfloats.getmember(3) << '\n';
 return 0;
}
100
3.1416
```

المساحات Namespaces:

تسمح المساحات بتجميع الاصناف او الاهداف او الاقترانات في اسم واحد وتنفيذ عملية الاستدعاء بطرق مختلفة وكما هو مبين في المثال التالي:

```
namespace myNamespace
{
 int a, b;
}
// namespaces
#include <iostream>
using namespace std;

namespace first
{
 int var = 5;
}

namespace second
{
 double var = 3.1416;
}

int main () {
 cout << first::var << endl;
 cout << second::var << endl;
 return 0;
}
5
3.1416
```

هذا ويمكن استخدام الكلمة المحجوزة using لتحديد عملية الاختيار وكما هو مبين في الامثلة التالية:

```
// using
#include <iostream>
```

```
using namespace std;

namespace first
{
 int x = 5;
 int y = 10;
}

namespace second
{
 double x = 3.1416;
 double y = 2.7183;
}

int main () {
 using first::x;
 using second::y;
 cout << x << endl;
 cout << y << endl;
 cout << first::y << endl;
 cout << second::x << endl;
 return 0;
}
```

```
5
2.7183
10
3.1416
```

```
// using
#include <iostream>
using namespace std;

namespace first
{
 int x = 5;
```

```
 int y = 10;
}

namespace second
{
 double x = 3.1416;
 double y = 2.7183;
}

int main () {
 using namespace first;
 cout << x << endl;
 cout << y << endl;
 cout << second::x << endl;
 cout << second::y << endl;
 return 0;
}
5
10
3.1416
2.7183
```

```
// using namespace example
#include <iostream>
using namespace std;

namespace first
{
 int x = 5;
}

namespace second
{
 double x = 3.1416;
}

int main () {
```

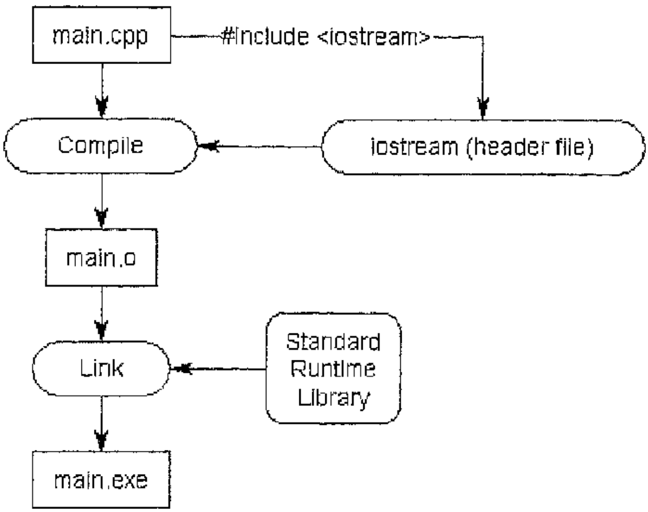
```
 {
  using namespace first;
  cout << x << endl;
 }
 {
  using namespace second;
  cout << x << endl;
 }
 return 0;
}
5
3.1416
```

تضمين الاقتران او الصنف في مكتبة سي بلس بلس Header files :

عندما يتعامل البرنامج مع عمليت الدخال والاخراج فننا نضمن المكتبة الخاصة بعمليات الادخال في البرنامج وكما هو مبين في المثال التالي:

```
#include <iostream>
int main()
{
  using namespace std;
  cout << "Hello, world!" << endl;
  return 0;
}
```

وعليه وبعد ترجمة البرنامج فانه يتم ربط المكتبة بالبرنامج وكما هو مبين في الشمل التالي:



لنفرض اننا نريد استخدام الاقتران التالي ضمن مكتبة سي بلس بلس:

```
int add(int x, int y)
{
   return x + y;
}
```

لعمل هذا لا بد من تعريف الاقتران بالشكل التالي وحفظه:

add.h

```
#ifndef ADD_H
   #define ADD_H

   int add(int x, int y) // function prototype for add.h
   {
```

```
    return x + y;  // without this the function doesn't know what
you want it to do
  }

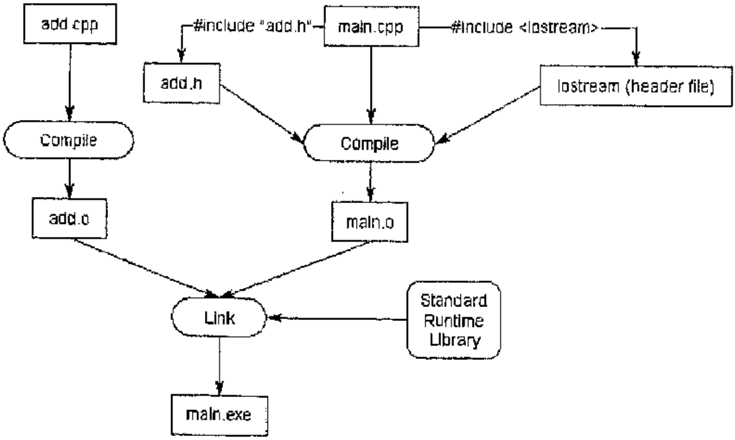
  #endif
```

وبعد حفظ هذا الاقتران فإنه يمكن استخدامه من البرنامج الرئيسي من خلال تضمينه بالبرنامج كما يلي:

```
#include <iostream>
#include "add.h" // this brings in the declaration for add()

int main()
{
  using namespace std;
  cout << "The sum of 3 and 4 is " << add(3, 4) << endl;
  return 0;
}
```

وسوف تتم عملية ربط الاقتران بعد ترجمة البرنامج الرئيسي وكما هو موضح في الشكل التالي:



والمثال التالي يبين كيفية تضمين الصنف في برنامج سي بلس بلس:

```
class Date
{
private:
   int m_nMonth;
   int m_nDay;
   int m_nYear;

   Date() { } // private default constructor

public:
   Date(int nMonth, int nDay, int nYear);

   void SetDate(int nMonth, int nDay, int nYear);

   int GetMonth() { return m_nMonth; }
   int GetDay()  { return m_nDay; }
   int GetYear() { return m_nYear; }
};

// Date constructor
Date::Date(int nMonth, int nDay, int nYear)
{
   SetDate(nMonth, nDay, nYear);
}

// Date member function
void Date::SetDate(int nMonth, int nDay, int nYear)
{
   m_nMonth = nMonth;
   m_nDay = nDay;
   m_nYear = nYear;
}
```

Date.h:

```
#ifndef DATE_H
#define DATE_H

class Date
{
private:
   int m_nMonth;
   int m_nDay;
   int m_nYear;

   Date() { } // private default constructor

public:
   Date(int nMonth, int nDay, int nYear);

   void SetDate(int nMonth, int nDay, int nYear);

   int GetMonth() { return m_nMonth; }
   int GetDay()  { return m_nDay; }
   int GetYear() { return m_nYear; }
};

#endif
```

Date.cpp:

```
#include "Date.h"

// Date constructor
Date::Date(int nMonth, int nDay, int nYear)
{
   SetDate(nMonth, nDay, nYear);
}

// Date member function
```

```
void Datc::SetDate(int nMonth, int nDay, int nYear)
{
   m_nMonth = nMonth;
   m_nDay = nDay;
   m_nYear = nYear;
}
```

المراجع

REFERENCES

The C++ Programming Language, 3rd Ed (Stroustrup, 1999) -- Every serious C++ programmer should have this book. It contains intermediate to advanced material, and covers both the language and the new standard libraries. Read chapters 2 & 3, then browse the rest of the book as you need it. The new special edition has two additional chapters, and I recommend getting that one if you can, but if you can't those chapters are also available on Bjarne Stroustrup's web site.**Highly recommended.**

Generic Programming and the STL: Using and Extending the C++ Standard Template Library (Austern, 1999) -- The best STL book I have found yet. The first few chapters are a pretty good introduction to the STL, and the bulk of the book is an excellent reference. Note that this covers the material from a very rigorous, almost mathematical point of view; you may want to get another book (such as Josuttis) for initial learning. **Highly recommended.**

The C++ Standard Library: A Tutorial and Reference (Josuttis, 2000) -- The only book so far that covers the new Standard C++ Library. This focuses specifically on the library itself rather than the C++ language. It is an excellent book for learning about all the standard library facilities. **Highly recommended.**

C++ Primer, 3rd Ed (Lippman and Lajoie, 1998) -- A very complete book at over 1200 pages, it includes tutorials on all aspects of the modern C++ language and standard library. Recommended if you want to learn C++ for the first time, and have the time to devote to going through the tutorials.

Essential C++ (Lippman, 2000) -- "C++ Primer Lite". This is the book to get if you have to get up and running with C++ as soon as possible, and need to learn on the job (in other words, most programmers). Be sure to follow this up with some of the more

extensive C++ books if you plan to continue using C++ professionally.

Magazine: The C/C++ User's Journal -- Every C++ programmer should have a subscription to this magazine. The magazine is devoted to C and C++ articles, with occasional Java thrown in. The emphasis is on practical programming techniques. **Highly recommended.**

Advanced C++

Exceptional C++ (Sutter, 2000) -- An investigation into good C++ programming strategies and styles, in the form of engineering puzzles. An good format for testing yourself; this book originated in an ongoing series of Usenet postings called "Guru of the Month" which appear in comp.lang.c++ regularly. Includes more than just the back postings (which are available on deja.com or at Herb Sutter's web site). See the ACCU review for details; if you use C++ much at all, you should have this book on your shelf. **Highly recommended.**

Standard C++ IOStreams and Locales (Langer and Kreft, 2000) -- An excellent book on the details of IOStream and i18n programming; the *only* book I know of that covers the new, standard IOStreams. You need this book if you're creating new stream or streambuf classes using the new standard, or if you want to take advantage of the i18n capabilities of the Standard C++ Library.

Generative Programming: Methods, Tools, and Applications (Czarnecki and Eisenecker, 2000) -- A possibly groundbreaking book which touches on techniques of generic programming as well as a host of other subjects. Definitely an advanced book, but well written. (No ACCU review yet.)

Classic C++ Books

Advanced C++ Programming Styles and Idioms (Coplien, 1992) -- A classic book on advanced C++ programming

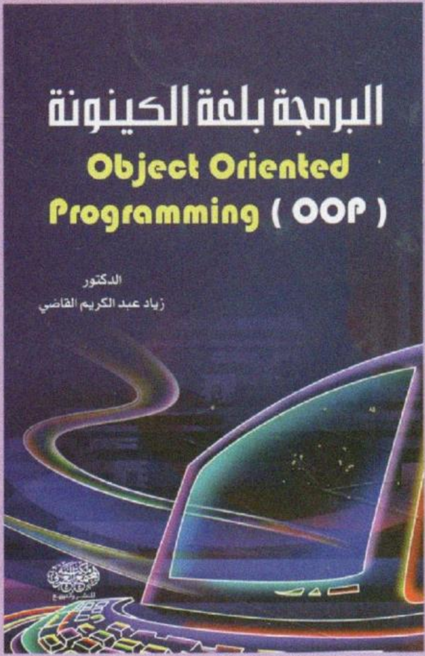
techniques. It predates the pattern movement, but it really is a collection of language-level patterns.

Ruminations on C++ (Koenig and Moo, 1996) -- Contains advanced C++ programming techniques. Some of them are now part of the standard library (iterators, generic programming). A good book to get after you read Coplien and Meyers.

The Design and Evolution of C++ (Stroustrup, 1998) -- Not a programming book, but a good background and history of how C++ came to be what it is today. If you are interested in why the language is the way it is, this is the book to read.

البرمجة بلغة الكينونة

Object Oriented Programming (OOP)



9789957831165

مكتبة المجتمع العربي للنشر والتوزيع
الأردن-عمان-وسط البلد- ش. السلط - مجمع الفحيص التجاري- تلفاكس، 2739 463 6 962+
خلوي 5651920 79 962+ ص.ب 8244 الرمز البريدي 11121 جبل الحسين الشرقي
الأردن - عمان - الجامعة الأردنية - ش. الملكة رانيا العبدالله - مقابل كلية الزراعة - مجمع زهدي حصوة التجاري
www.muj-arabi-pub.com
E-mail:Moj_pub@hotmail.com